Multistage Adaptive Load Balancing for Big Active Data Publish Subscribe Systems

Hang Nguyen*
hangn5@uci.edu
University of California, Irvine
Irvine, California, United States

Md Yusuf Sarwar Uddin
msarwaru@uci.edu
University of California, Irvine
Irvine, California, United States

Nalini Venkatasubramanian
nalini@ics.uci.edu
University of California, Irvine
Irvine, California, United States

ABSTRACT

In this paper, we address issues in the design and operation of a Big Active Data Publish Subscribe (BAD Pub/Sub) systems to enable the next generation of enriched notification systems that can scale to societal levels. The proposed BAD Pub/Sub system will aim to ingest massive amounts of data from heterogeneous publishers and sources and deliver customized, enriched notifications to end users (subscribers) that express interests in these data items via parameterized channels. To support scalability, we employ a hierarchical architecture that combines a back-end big data cluster (to receive publications and data feeds, store data and process subscriptions) with a client-facing distributed broker network that manages user subscriptions and scales the delivery process. A key aspect of our broker capacity is its ability to aggregate subscriptions from end users to immensely reduce the end to end overheads and loads. The skewed distribution of subscribers, their interests and the dynamic nature of societal scale publications, create load imbalance in the distributed broker network. We mathematically formulate the notion of broker load in this setting and derive an optimization problem to minimize the maximum load (an NP-hard problem). We propose a staged approach for broker load balancing that executes in multiple stages — *initial placement* of brokers to subscribers, *dynamic subscriber migration* during operation to handle transient and instantaneous loads and occasional *shuffles* to re-stabilize the system. We develop a prototype implementation of our staged load balancing on a real BAD Pub/Sub testbed (multinode cluster) with a distributed broker network and conduct experiments using real world workloads. We further evaluate our schemes via a detailed simulation studies.

CCS CONCEPTS

• **Software and its engineering → Publish-subscribe / event-based architectures**.

ACM Reference Format:
Hang Nguyen, Md Yusuf Sarwar Uddin, and Nalini Venkatasubramanian. 2019. Multistage Adaptive Load Balancing for Big Active Data Publish Subscribe Systems. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329508

1 INTRODUCTION

The distributed publish/subscribe paradigm is a popular communication architecture for information dissemination; here publishers publish messages to a broker network that matches and routes these messages to interested subscribers. In this paper, we propose a novel architecture, BAD Pub/Sub [5], to significantly scale both the volume of data and subscribers while enabling customized, enriched notifications to subscribers using a big data management backend. The BAD Pub/Sub approach aims to leverage the benefits of both Big Data Management Systems (BDMS) and distributed broker Pub/Sub systems while overcoming key limitations. To address the scale issue, our BAD Pub/Sub system combines BDMS for data collection and ingestion with a distributed broker network for information dissemination to reach a very large number of end users rapidly [2–4].

An interesting aspect of the proposed BAD Pub/Sub approach lies in our ability to create enriched notifications/reports for subscribers by combining information in publications with external data sources. In contrast to traditional publish/subscribe systems where publications from a single publisher are routed as is to subscribers, notifications in BAD Pub/Sub are produced against a set of datasets - each dataset is one stream of publications, and enriched with pre-loaded data to create comprehensive reports for subscribers. With the ability to store incoming streams of publications, match and enrich them at a big data backend, BAD Pub/Sub make notifications persistent.

In traditional BDMS [3], users gain information via explicit queries to a database system; users requiring continuous updates from the system must mimic the notion of subscriptions using DB trigger functionalities (slow); support for continuous queries are available in big stream management systems, however data in such systems are volatile. In a BAD Pub/Sub system, users register their queries as subscriptions to topics (channels) of interest and notifications/results are automatically generated and delivered to users when there are updates which match their interests. BAD Pub/Sub systems inherently support sharing; results for subscriptions shared across subscribers can be produced once and delivered to all subscribers. The ability to perform such subscription aggregation is our first step in supporting system scaleup. To further support scalability, BAD Pub/Sub employs a hierarchical architecture with three layers (Figure 1). Information from multiple data sources (publishers) are fed into different datasets in the data cluster at the top layer. The end users/subscribers are represented at the lowermost layer. The broker network in the middle layer manages subscribers and their subscriptions, aggregates/subsumes subscriptions from multiple subscribers and passes them to the data cluster. The broker network also retrieves results/notifications from the data cluster and disseminates results to matched subscribers.

While BAD Pub/Sub presents desirable features for scalability, ensuring performance and robust operation under dynamic conditions is challenging. Non-uniform distribution of subscribers and the dynamic nature of publications lead to an unbalanced load distribution among brokers - this affects system performance and the ability to disseminate notifications/results to all matched subscribers in a timely manner. Quantification of broker load meaningfully must incorporate notions of management loads and communication overheads. The management loads at brokers account for the subscribers and subscriptions management while the communication overheads capture the steps involve in (a) retrieving notifications/results for all subscriptions from the backend data cluster and (b) disseminating notifications/results from the broker to all matched subscribers. The dynamic usage pattern of subscribers in term of the number and the nature of subscriptions that each subscriber generates, the unpredictable states of subscribers (active vs. inactive), make the effective assignment of subscribers to brokers not a trivial task.

To cope with different characteristics of the system, in this paper, we propose a set of adaptive load balancing schemes: i) *initial placement* - here, we initially assign a broker to a subscriber after the subscriber registers itself with the system to start service; policies explored include geo-location based allocation, round robin and random broker selection. The next step is the *dynamic subscriber migration* step where we dynamically migrate a subset of subscribers from highly loaded brokers to lightly loaded ones to handle the fluctuation of broker load distribution during the course of operation. To address the extreme load imbalance, we introduce a *shuffle* policy that re-configures the entire system to optimally redistribute subscribers among brokers.

The following are key contributions of this paper: i) We propose a practical and efficient multistage load balancing framework for BAD Pub/Sub (with initial assignment, dynamic migration and shuffle) - Sec. 3; ii) We develop a mathematical model for broker load in BAD Pub/Sub and formulate the associated load balancing problem as an NP hard problem - Sec. 4 ; iii) We design algorithms for each stage of the load balancing framework that are dynamic migration and shuffle - Sec 5; iv) Implement the proposed techniques in a BAD Pub/Sub prototype system developed over a local cluster of machines using real world use cases; v) Evaluate performance of different combination of load balancing policies with real world use cases on the prototype platform and simulation experiment - Sec 6.

2 RELATED WORK

Load balancing has been a well researched topic since the introduction of parallel and distributed computing and been largely applied in many distributed contexts from distributed databases to high performance computing systems. Sample efforts include key-value pair assignment in distributed networked cache systems [16], request balancing in crowd-sourced CDNs [17], virtual machine assignment in cloud computing [18], object distribution among peer nodes in P2P systems [19], sensor partitioning into clusters in WSNs [20, 21], event key-grouping in complex event processing (CEP) engines [22]. etc. Overall, load balanced systems can be achieved via smart objects/tasks distribution strategies initially and migration techniques if needed in response to the system load flexibility afterwards [19, 22, 24, 25].

Load balancing in the context of publish/subscribe has been studied in both content-based [10, 23] and topic-based [6] systems using different architectures, where distributed hash table (DHT)-based, tree-based, cluster-based and community-based approaches have been used to organize the broker network. Many of the approaches to load balancing in Pub/Sub aim to divide the overheads of subscription management (maintenance, matching) and publication processing (routing) among distributed brokers by (a) partitioning topic/subscription space across brokers through hashing [31–33] and (b) clustering techniques [8, 10, 12]. In consistent hashing, each broker node in the network has an unique identifier. The topic is hashed to the same domain as the identifier space and each topic will be assigned to the closest virtual identifier. With efficient placement of broker nodes in the virtual space, each broker node will be responsible for an equal share of topics. In clustering techniques, for example in [8], event/publication space is organized into partitions. The popularity of the publication determines the number of clusters for each partition - this allows balancing the subscription maintenance load among clusters while publication forwarding load is balanced among brokers in the same cluster. In [10], brokers are partitioned into clusters. Each cluster contains one cluster head who serves only the publishers, and a set of edge brokers who serve subscribers. Clusters are organized into a hierarchical architecture to allow two levels of load balancing: local load balancing among brokers within the cluster and global load balancing among cluster heads of different clusters. Here, bit vectors are used to profile subscription load and offloading algorithms determine an appropriate set of subscriptions for migration to balance multiple performance metrics of a broker including input rate, output rate and matching rate. In [12], the author exploits similarity for clustering brokers then uses offloading mechanism for inter-community load balancing and uses filter replication for intra-community load balancing.

In contrast, the data cluster in our BAD Pub/Sub approach handles a bulk of the subscription matching and storage tasks, broker workload primarily involves communication with subscribers and the data cluster to manage throughput. Unlike [6] that involves high-cost migrations of the partitions corresponding to a topic, we perform subscriber migrations that are relatively lightweight (i.e. move subscriber connection with minimal state migration). The ability of the broker network to aggregate subscriptions and the feature of our BAD Pub/Sub systems which are able to persist publications, subscriptions, and notifications in the back-end data cluster change the nature of the load balancing problem for BAD Pub/Sub as compared to existing Pub/Sub systems.

3 BAD PUB/SUB ARCHITECTURE AND LOAD BALANCING APPROACH IN BAD PUB/SUB

In this section, we outline the architecture of a BAD Pub/Sub system with all of its components and interactions. We also describe the characterization of load in the context of a BAD Pub/Sub system along with our approaches to balancing load in the system.

3.1 BAD Pub/Sub Architecture

Figure 1 illustrates the hierarchical architecture of a BAD Pub/Sub system, which may have three layers: i) the back-end data cluster, ii) front-end subscribers, and iii) a set of brokers in the middle. The data cluster is responsible for ingestion, storage and management of incoming data (e.g., *publications* from publishers), whereas the broker network is for subscription management and notification delivery to the end subscribers [4]. The data cluster runs the extended existing open-source BDMS, Apache AsterixDB [35] that provides the necessary data storage and query processing functionality required for the BAD Pub/Sub system. The data cluster allows different data sources (i.e., publishers) to feed their publications into distinct *datasets* on which queries can be built using standard SQL-like query language. Instead of subscribing to a topic as is done in topic-based pub-sub systems, in BAD Pub/Sub, subscribers subscribe to *functions* that can match data from more than one sources (i.e., datasets) and generate notifications accordingly. We refer to these functions as *channels*. More specifically, channels are actually *parameterized* queries defined on top of the Apache AsterixDB hosted by the data cluster such that subscribers can register their interests through specifying values for those parameters in their subscriptions. Each channel has a qualified unique name and the underlying query is written using the corresponding DML (Data Manipulation Language) supported by AsterixDB .

Subscriptions to channels. Each subscriber attaches itself to a broker (how to attach to a broker is discussed later) and passes its subscriptions to that broker. The broker then passes these subscriptions back to the data cluster and pulls the results for these subscriptions when they are generated. Each subscription is specified by a channel name (the name of the channel to which the subscription is intended for) and the values to the parameters of that channel (if any). The same set of parameter values passed for the same channel constitutes identical subscriptions (even though they may come from different subscribers). If a broker receives identical subscriptions from multiple subscribers, it issues only one subscription to the back-end data cluster and results are shared among those subscribers having those identical subscriptions. This is we referred to as *subscription sharing*. The benefit of this sharing is that the broker can retrieve one set of results (notifications) for all sharing subscriptions and push the same results onto different subscribers. Along the same direction, the data cluster can potentially receive identical subscriptions from multiple brokers, in which it does not keep each of them, rather maintain one subscription in the data cluster and deliver notifications to the respective brokers. In this way, the number of unique subscriptions that the backend data cluster maintains (which we call *backend* (BE) subscriptions) can be far smaller than the number of original subscriptions (which are called *frontend* (FE) subscriptions).

Each channel is executed periodically by the data cluster against the subscriptions it holds. The period of a channel is usually specified when the channel is created and the channel query gets executed at that interval. At each channel execution, the underlying query is run on a set of associated datasets in the data cluster (as

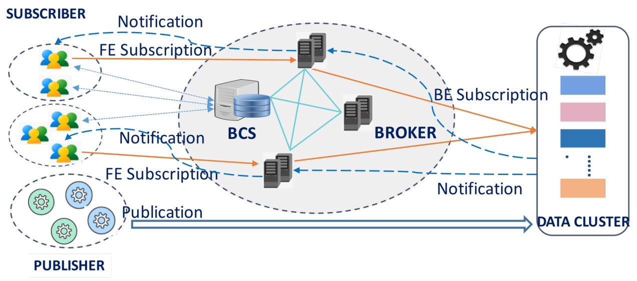

Figure 1: Architecture of a BAD Pub/Sub system

per the channel query code) and generates new results (matched publications from those datasets) against all BE subscriptions for that channel. These results are stored in a designated result dataset in the data cluster and the corresponding brokers with matched BE subscriptions are notified. The brokers then issue pull requests to fetch the set of result records for each matched BE subscription and deliver notifications to the matched subscribers.

The fact that each channel actively executes periodically to produce new results for registered FE subscriptions from subscribers without them having to send queries explicitly every single time and results produced for a single BE subscription can be shared among multiple subscribers who have sharing subscriptions helps to save resource effectively and scales up the system to support a very large number of subscribers. The subscribers in our system can be in online vs offline state in the sense that their connections to the assigned brokers are active or not. During offline period, the subscriber stops receiving notifications/results for its subscriptions.

Broker Coordination Server. To facilitate subscribers to attach to a broker and to coordinate operations and states among the brokers, the BAD Pub/Sub systems can have a Broker Coordination Server (BCS), which can act as the public end-point of the broker network. Any incoming subscriber first registers itself with the BCS and asks for a "suitable" broker for itself. The subscriber then connects to the corresponding broker. Incoming brokers also register themselves with the BCS and all brokers periodically pass their load statistics to the BCS so that the BCS can trigger different load balancing techniques in order to keep a uniform load distribution among brokers, which calls for the load balancing problem in BAD Pub/Sub that we describe next.

3.2 Load Balancing in BAD Pub/Sub

The load balancing (LB) problem in BAD Pub/Sub is attributed to distributing near-equal "load" across all the brokers in the system. Unbalanced load distribution among the brokers can arise because of the skewed user distribution, the unpredictable usage pattern of subscribers in terms of the number of subscriptions that each subscriber can create over time, the nature of those subscriptions, and the unprecedented data volume ingested in the data cluster. Skewed load distribution among the brokers arguably degrades the performance of the whole system in terms of the number of subscribers that the system is able to support and the end-to-end latency introduced by the broker network in delivering results from the data cluster to the subscribers. In order to scale up the system and to be able to support a very large number of subscribers, we need to have an effective mechanism that equally distributes the load among the brokers so that no broker has to process an excessive amount of load compared to others. This would avoid the unbalanced workload distribution among brokers where some subset of subscribers under the service of overloaded brokers would endue longer latency in receiving their notifications while other under-loaded brokers have redundant resources.

Definition of broker load: The task of load balancing in the context of BAD Pub/Sub systems is challenging because here brokers cannot look at a single or static attribute, such as the number of subscribers or the number of subscriptions, to balance load. Instead, the load at a broker is a composite metric that depends on many factors, such as the number of subscribers that the broker is serving (as the broker needs to maintain connections with all of its subscribers) as well as the set of subscriptions from each of those subscribers and the data volume generated for each subscription itself from the data cluster per channel execution and the frequency of the channels. Because of the diversity of subscriptions coming from subscribers and the overlap of subscriptions among them (subscription sharing), the volume of data (both incoming and outgoing data) plays a big role in assessing the overall load at a broker. Therefore, we define the load of a broker as the total volume of data the broker needs to handle per unit of time. This total volume of data accounts for both the amount of data retrieved from the data cluster (for all BE subscriptions that the broker maintains with the data cluster) and the amount of data disseminated towards the subscribers (notifications delivery for all FE subscriptions). This definition is consistent with other relevant work in the literature [6, 10].

3.3 Our Load Balancing Approach

Because of a wide range of load fluctuation that each broker may experience over time, we need to constantly monitor the load status of the system to take timely actions in order to balance load in the broker network. As per operation, all brokers in the system send their load updates to the BCS periodically. This helps the BCS to maintain the global view about the system's overall load across the brokers and to detect if load is skewed across brokers (based on a quantitative metric we define later on). We take a multistage approach in balancing the load across the brokers in the sense that the system invokes a set of techniques, three to be precise, at different temporal granularity. The first technique is *initial placement* that assigns an incoming subscriber to an existing broker. Usually, this initial placement may be subscription-agnostic because the subscriber may not have any subscriptions when they join (subscriptions originate later on). Because of the dynamic volume of data generation from the data cluster, the matching rate of each individual subscription, the changing set of subscriptions that each subscriber make over time, any initial placement can lead to a skewed load across the brokers in the future. Therefore, we propose two more techniques, namely *dynamic migration* and *shuffle* to fix the unbalanced condition depending on how much variation of load among the brokers is detected. Figure 2 demonstrates our overall load balancing approach.

Dynamic migration refers to moving one or more active subscribers from their current brokers to new brokers. The key task

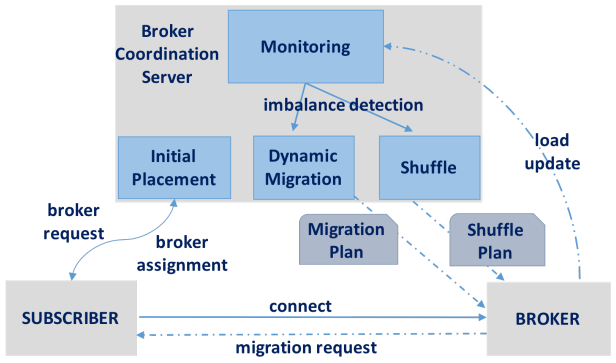

Figure 2: Multistage Adaptive Load Balancing Framework

here is to decide who moves to where. The BCS takes these decisions and let them to be realized by the brokers and tries to invoke those migrations only when needed. The basic idea is to move the heaviest subscribers from the heaviest brokers to less loaded brokers and keep doing this until the system reaches to a balanced condition. However, when the system experiences an extreme load imbalance, in order to quickly fix the bad situation and further optimize the distribution of subscribers among the brokers, the system invokes *shuffle*, which redistributes the whole set of current active subscribers over all brokers (overhauling the entire subscriber-broker assignment from the scratch). While the dynamic migration can be invoked in response to a slight load imbalance, invoking the shuffle is rare and only happen when the system experiences an extremely unbalanced load distribution.

4 SYSTEM MODEL AND PROBLEM FORMULATION

Let us consider the system has a fixed (m) number of brokers $B = \{j : 1, 2, \ldots, m\}$ at any instance of time, which are all registered with the BCS. Let there be a collection of n users in the system $U = \{i : 1, 2, \ldots, n\}$ who generated a total of q BE subscriptions (the set of unique subscriptions held at the data cluster). Since our BAD Pub/Sub system is instrumented to deliver notifications to only online users, we assume these users are online and are currently active to receive notifications. The set of all BE subscriptions is denoted as $S = \{k : 1, 2, \ldots, q\}$. Note that each BE subscription can potentially be originated from multiple brokers and initiated from multiple subscribers at those brokers. It is also worth noting that while FE subscriptions are specific to subscribers, BE subscriptions are independent of both subscribers and brokers. A broker can have a BE subscription held on it only if it has at least one FE subscription attached to it. In the following, if not otherwise stated, we use i, j and k to denote a subscriber, a broker and a BE subscription, respectively.

Let y_{ik} denote a binary indicator if subscriber i has a FE subscription that can be attached to BE subscription k. Let z_{jk} denote a binary indicator if broker j has to maintain a subscription to BE subscription k. As we have said, this only happens if the broker has at least one FE subscription for k originated from some subscriber connected to that broker. Let x_{ij} denote the current subscriber-to-broker assignment in the system. That is, x_{ij} is set of 1 if subscriber i is attached to broker j. Actually, this assignment metric $X = \{x_{ij}\}$ is the one that the system needs to compute at a certain time. This assignment is done by the BCS.

Now let us consider how the load is constituted at each broker. Note that, by definition, the load of a broker is the sum of total incoming data and the total amount of outgoing data per unit of time. For each BE subscription at a broker, the broker needs to pull the results from the data cluster each time new results are populated against the associated channel and pushes the same results onto the subscribers who have FE subscriptions attached to that BE subscription. Let λ_k be the data rate at which new results are generated for BE subscription k. Hence, the broker load, denoted as F_j for broker j, has two parts: incoming data volume I_j and outgoing data volume O_j.

The incoming load can be obtained as:

$$I_j = \sum_{k=1}^{q} z_{jk} \times \lambda_k \tag{1}$$

where z_{jk} (if broker j has a subscription to k) is given by:

$$z_{jk} = 1 - \prod_{i=1}^{n} \left(1 - x_{ij} \times y_{ik}\right) \tag{2}$$

Let n_{jk} be the number of total FE subscriptions attached a BE subscription k at broker j, which is given by:

$$n_{jk} = \sum_{i=1}^{n} x_{ij} \times y_{ik} \tag{3}$$

Therefore, the total volume of data delivered per unit of time to the attached subscribers from broker j is:

$$O_j = \sum_{k=1}^{q} n_{jk} \times \lambda_k \tag{4}$$

Combining the above two terms, we obtain the total load of broker j as:

$$F_j = I_j + O_j = \sum_{k=1}^{q} z_{jk} \left(1 + n_{jk}\right) \lambda_k \tag{5}$$

As we can see the first part of the load depends on the BE subscriptions that a broker maintains (which in turn depends on the degree of shared subscriptions, the more the sharing the less the incoming load), whereas the second part is directly attributed to notification delivery to attached subscribers. This sharing among subscriptions adds a non-trivial complexity to the load balancing problem, which is considered in our solution strategies.

We formulate the load balancing problem as a *minmax* problem, that is, the objective of the load balancing problem is to compute the assignment metric $X = \{x_{ij}\}$ so as to minimize the maximum load across all brokers. One can think of other forms of objective functions to balance load, such as minimizing the difference between the max load and the min load, or minimizing the variance of load, etc. We assume that these choices are orthogonal to the problem at hand and can all be good candidates to check as none of them essentially raises or eases the complexity of the underlying problem. Having said that we define our objective of load balancing is to minimize the maximum load. More formally:

Given:

$$R = \{\lambda_k : k = 1, \ldots, q\}$$
$$Y = \{y_{ik} : i = 1, \ldots, n; k = 1, \ldots, q\}$$

Find $X = \{x_{ij} : i = 1, \ldots, n; j = 1, \ldots, m\}$ so as to

$$\min \max_j F_j$$

subject to:

$$\sum_{j=1}^{m} x_{ij} = 1, \forall i = 1, \ldots, n$$

The constraint indicates that at any given time each subscriber can be attached to exactly one broker. The above problem is NP-Hard, which can be reduced to the Multi-Processor Scheduling (MPS) problem.

Reduction to the Multi-Processor Scheduling problem. The optimization problem we examine can be seen as an instance of the MPS optimization problem which is a well-known NP-complete problem [34]. In MPS problem, there are a set of m identical machines and n jobs. Each job has a processing time $p_i \geq 0, \forall i \in n$ and the optimization statement is to assign jobs to machines so as to minimize the maximum processing time among all machines. Our optimization problem can be reduced to the MPS problem where brokers are machines and subscribers are jobs. The MPS problem is a special case of our optimization problem. In more detail, when there is no subscription sharing among subscribers at all in the whole system, says, each subscriber make a unique subscription and each unique subscription has a specific load (specific data rate), then finding the allocation of subscribers among available brokers which generates the most balanced load distribution is exactly the MPS problem.

Since the problem is NP-Hard, we devise algorithms based on greedy heuristics that we describe in the next section. The key idea is to iteratively choose one subscriber at a time (the heaviest one) from from the most loaded broker and assign it to the lightest broker. We observe that when a subscriber is picked to migrate, the additional load endued by the destination broker equals to the sum of λ's of the subscriptions that the subscriber has no matter which destination broker is chosen. On the other hand, the change in incoming load at the destination broker depends on the subscription commonality between the subscriber and the broker itself as the broker only needs to retrieve additional amount of result data for those new subscriptions from the subscriber which are not currently held by the broker. This is the key insight in developing the heuristic algorithms we present in Algorithm 1.

A question remains when to invoke this subscriber to broker re-assignment. Ideally, the re-assignment happens when the system detects an "unbalanced" state from the existing assignment. The BCS keeps track of loads across all brokers and triggers re-assignment when needed (we refer to this as dynamic migration). The BCS uses the coefficient of variation (*cov*), the ratio of the standard deviation to the mean of broker loads, as an indicator of the degree of load imbalance across brokers. The *cov* is calculated as follows:

$$cov = \frac{\sigma}{\mu} \tag{6}$$

where

$$\sigma = \sqrt{\frac{\sum_{j=1}^{m}(F_j - \mu)^2}{m}} \tag{7}$$

$$\mu = \frac{\sum_{j=1}^{m} F_j}{m} \tag{8}$$

A low *cov* value indicates a balanced system whereas the higher value indicates an unbalanced one. The BCS uses a threshold α on *cov* to determine when to trigger the dynamic migration.

5 LOAD BALANCING POLICIES

Our load balancing framework has three phases each of which addresses different sub-problems of the system over the course of operation. These three phases are: i) initial placement, ii) dynamic migration, and iii) shuffle.

5.1 Stage 1: Initial Placement

The initial placement is based on different criteria to assign a suitable broker to a newly joined subscriber without knowing apriori the subscriptions of the subscriber. We explore three policies for the initial placement and the mapping of subscribers to brokers.

Nearest Broker (NR) Assignment As the name suggests, when a subscriber sends a request for a broker to the BCS, the subscriber also attaches its current location information to allow BCS return the nearest broker to the subscriber. The distance can be, however, the geographic distance or the network latency (the choice can be implementation dependent). Since the broker network is geo-distributed over a very large area, by assigning the nearest brokers to subscribers the system aims to minimize the latency of results delivery from brokers to subscribers. However, the skewed distribution of subscribers in space may lead to a skewed mapping of subscribers to brokers and can degrade the latency experienced by the subscribers.

Round Robin (RR) Placement The round robin method, on the other hand, balances the number of subscribers assigned to each broker, ideally each broker serves an equal number of subscribers. Since the load of brokers depends on other factors as well other than only the number of subscribers, such as the number of subscriptions and the nature of those subscriptions, round robin placement also can not guarantee the balanced load distribution.

Random (RND) Placement In this scheme, the BCS assigns an arbitrary broker to each subscriber. If subscribers have a uniform subscription distribution over the subscription space, the random assignment can produce a temporally balanced system but it can not be guaranteed.

People may argue that the Least Loaded (LL) Broker Placement should be a good policy to explore. However, the LL placement turns out not to perform well, especially when many empty subscribers come to the system at similar time and they are all assigned to the current least loaded broker which will shortly overwhelm the assigned broker as those subscribers start producing subscriptions.

Algorithm 1 Multistage adaptive load balancing

```
                                  Input:
$U = \{i : 1, .., n\}$                                       ▹ subscriber set
$B = \{j : 1, .., m\}$                                           ▹ broker set
$R = \{\lambda_k : k = 1..q\}$                        ▹ subscription data rate
$X = \{x_{ij}, i = 1..n, j = 1..m\}$         ▹ subscriber broker assignment
$Y = \{y_{ik}, i = 1..n, k = 1..p\}$      ▹ subscriber subscription matrix
                                  Output:
$M = \{\}$                                                ▹ migration plan
1:  $scheme \leftarrow$ {LDM, SDM}
2:  if $cov > \gamma$ & $\mu > \theta$ then
3:      $M \leftarrow$ SHUFFLE
4:  end if
5:  if $cov > \alpha$ & $\mu > \beta$ then
6:      while $cov > \alpha$ & $\mu > \beta$ do
7:          $b \leftarrow \arg\max_{j \in B} F_j$              ▹ broker of maximum load
                                          ▹ loads of subscribers at $b$
8:          $U_b = \{u_i : u_i = \sum_{k=1}^{p} y_{ik} \times \lambda_k, \forall i \in U, x_{ib} = 1\}$
9:          for $u_i$ in $Sorted(U_b, reverse = True)$ do
                       ▹ similarity between subscriber $i$ and broker $j$
10:             $simTable = \{sim_j : sim_j = \sum_{y_{ik}=1, z_{jk}=1} \lambda_k, j \in B\}$
11:             if $scheme$ == LDM then
12:                 $b' \leftarrow \arg\min_j F_j$
13:             end if
14:             if $scheme$ == SDM then
15:                 $b' \leftarrow \arg\max_{j, F_j < \mu} simTable$
16:             end if
17:             if $\underset{x_{ib'}=1}{F_{b'}} \leq \underset{x_{ib}=1}{F_b}$ then
18:                 $M \leftarrow \{b : [i, b']\}$
19:                 break
20:             end if
21:         end for
22:         Update $F_b, F_{b'}$
23:         $x_{ib} \leftarrow 0, x_{ib'} \leftarrow 1$
24:     end while
25: end if
26: return $M$
```

5.2 Stage 2: Dynamic Migration (DM)

The initial placement of subscribers to brokers does not guarantee to produce a balanced system over time. Therefore, we design the second phase to fix the system configuration whenever a skewed load distribution is detected. As the BCS collects current loads of all brokers periodically every so often, it calculates the coefficient of variation *cov* as a measure of load imbalance across the brokers. If *cov* exceeds a threshold α and the mean load μ is greater than the lower bound (Algorithm 1), the BCS generates a migration plan and sends the plan to all target brokers to initiate the migrations. Leveraging the idea of the *Longest Processing Time* greedy algorithm to solve the MPS problem, we develop two strategies that are load based and similarity based DM as described in detail below. Our general intuition is to keep selecting a subscriber from the most loaded broker in each iteration as a candidate to migrate to a less loaded broker. By doing that, after every iteration, the maximum load of brokers is decreased or the number of brokers with maximum load (if there are more than one brokers with maximum load) is reduced until no further subscriber migration can be performed or the system reaches the balanced state.

Load-based Dynamic Migration (LDM): In this technique, the subscribers from the heaviest broker are ranked by their individual load. Then, each subscriber is checked for a valid migration in the order of decreasing load. A migration is said to be valid if after migration, the load of the destination broker does not exceed the original load of the source broker. For the LDM scheme, the destination broker is always the currently least loaded broker. For each valid migration, one entry is added to the migration plan that contains a list of tuples specifying which subscribers need to migrate from their current brokers to new brokers. The list is indexed by the source brokers and at the end of the algorithm (once the plan is populated), the respective brokers are notified with the list of subscribers to shred off and the destination brokers for those subscribers.

Similarity-based Dynamic Migration (SDM): In selecting the destination broker for a subscriber migration, different destination brokers will endure the same amount of additional outgoing load, but different destination brokers will have different amount of additional incoming load. Therefore, in SDM, our technique is to select the destination broker as the one that has the minimum increase of load when accepting the migrated subscriber. In other words, the migrated subscriber prefers to move to a broker that holds the largest subscription sharing with it. The degree of sharing between a subscriber and a broker is measured as the *subscription similarity score*, which is defined as the sum of data rates (λ's) of those shared subscriptions. However, in order to reduce the number of redundant migrations which involve the multiple migrations of a single subscriber back and forth among several brokers, the chosen destination broker should have the current load less than some threshold (maybe the average load of all brokers) in order to be able to accept a migrated subscriber.

5.3 Stage 3: Shuffle (SH)

The shuffle scheme is triggered when the system experiences an extreme skewed load distribution that means the current configuration is not appropriate. In order to optimally change the current assignment of subscribers to brokers, the shuffle process looks at the whole set of subscribers and their subscriptions to generate an near-optimal subscriber partition among the brokers which potentially produces an uniform load distribution across all brokers. We implement a simple greedy shuffle algorithm as presented below.

Greedy Shuffle Algorithm (GSH): In order to generate a near-optimal subscribers to brokers assignment, all brokers start with no subscribers. Subscribers are then ranked by their individual load measured as the total amount of data rates for their subscriptions. We iteratively assign the heaviest subscriber to the current least loaded broker and the load of the broker is then updated by the equation 5. The process finishes when all subscribers are assigned to brokers.

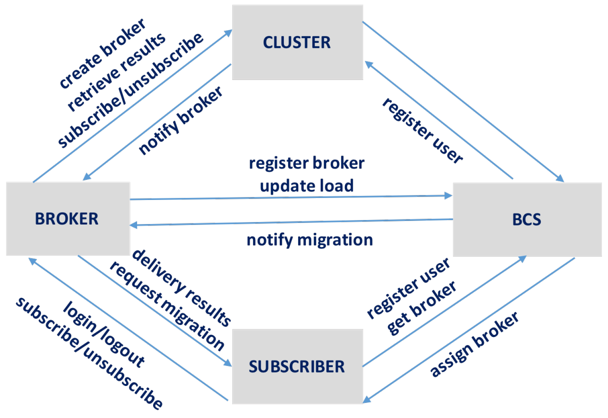

Figure 3: Interactions between different BAD Pub/Sub components

6 EXPERIMENTAL EVALUATION

In this section, we provide an experimental evaluation of our proposed multistage adaptive load balancing scheme under different settings in a real prototype implementation and in a simulation experiment.

6.1 Test-bed Evaluation

For the test-bed evaluation, we build a small scale prototype that includes all the components/functions of the BAD Pub/Sub systems.

Fig. 3 demonstrates in detail the interactions among these components.

6.1.1 Prototype Implementation. We describe in more detail the implementation of those main components in our BAD Pub/Sub systems as below:

Data Cluster: The data cluster, written in Java, leverages an existing open-source BDMS, Apache AsterixDB with additional features needed to build a BAD framework including: *data feeds* to allow rapid data ingestion from different data publishers into the system, *repetitive channels* to allow subscription registration from subscribers. We create an additional set of RESTful APIs to enable communication between the data cluster and brokers and the BCS, such as: *createbroker*, *createchannel*, *subscribe*, *unsubscribe*, etc.

BCS and Broker Servers: Our BCS and brokers, written in Python3, are RESTful HTTP servers built on top of the Tornado web framework. We develop a set of RESTful APIs in both BCS and broker servers to support communication among them and with subscribers. Several RESTful APIs in BCS are *registeruser*, *registerbroker*, *requestbroker* and in broker servers are *login*, *subscribe*, *unsubscribe*, *logout*, etc.

Subscriber: Subscribers are HTTP clients written in Python3.

6.1.2 Prototype Setup. Our prototype implementation consists of one data cluster, one BCS, 5 brokers and 400 subscribers. The data cluster consists of 4 Intel NUC nodes. Each node has an i7-5557U CPU processor (4 cores), 16 Gigabutes RAM and 1TB HD. The node are connected via a Gigabit Ethernet switch. The BCS and 5

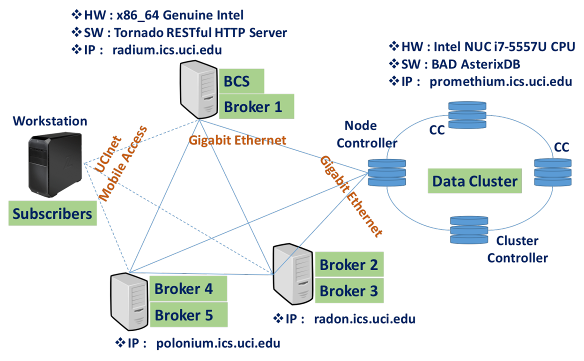

Figure 4: Prototype Implementation

brokers run on 3 x86_64 Genuine Intel machines (each equipped with 4 CPUs - Intel (R) Core(TM) i7-5557 CPU @3.10GHz, 16 GB RAM, 1TB HD) in an Ethernet LAN. Two machines run 2 broker server instances each and one machine runs one BCS instance and one broker server instance. The 400 subscribers connect to the system from a workstation machine and via the WiFi campus network. Figure 4 shows the detailed framework of our prototype implementation.

We build a hypothetical emergency notification application that creates three datasets in the data cluster: *UserLocations* storing records about the current locations of every subscribers, *EmergencyReports* containing records for each emergency report generated as emergencies occur, and *Shelters* holding known locations of about 200 emergency shelters. *UserLocations* and *EmergencyReports* will be continuously ingested into the data cluster while the *Shelters* will be loaded once and used as a static dataset. We model the realistic mobility of subscribers using the Opportunistic Network Environment (ONE) simulator [36]. We use the pre-built simulation of the city Helsinki in the ONE simulator to generate subscribers' movement in a simulated world. Subscribers' locations are updated to the data cluster every second via the broker network. Similarly, we also use the ONE simulator to generate emergencies. We create eight emergency publishers and allow them to traverse rapidly and randomly around the simulated Helsinki map. In total, we generate around 10,000 emergency reports of eight different types in the range (200, 700) bytes in size within 30 minutes to feed into the data cluster.

We create seven channels of different channel execution periods (10, 20 and 30 seconds) and different sets of parameters. The parameter set includes *Event Type*, *Event Location* and *User Location* to allow subscribers to specify the type of events, the location of events or simply any events happen near them in their subscriptions. We generate time series interactions of subscribers to the system including *login*, *logout*, *subscribe* and *unsubscribe* in a scenario file. We model each subscriber to create (1,3) subscriptions per channel and to subscribe to (1,5) channels in total. All subscribers login to the system in the first four minutes and make all subscriptions in the next four minutes for an experiment run. Overall, the system produces around 600 BE subscriptions and 2,200 FE subscriptions. We run the same experiment setup multiple times and apply different load balancing schemes to compare the performance of these schemes. When measuring the data rate of subscriptions and the load of subscribers and brokers, we use moving average measurements to avoid instant peaks.

6.1.3 Prototype Results. By applying our proposed load balancing techniques, we aim to minimize the maximum load of brokers and load imbalance represented by the coefficient of variation *cov* indicator. Therefore, we show the effectiveness of our schemes on these two performance metrics: *maximum load* and *cov*. The cost is measured as the number of subscriber migrations needed. We show the effect of both dynamic migration and shuffle schemes and demonstrate that our load balancing techniques work independent of the initial placement policies.

Comparison of integrated load balancing strategies: We study our integrated load balancing schemes on three placement policies including random, round robin and nearest broker and evaluate the performance of our schemes on these different policies. Fig. 5 and Fig. 7 present the effect of no load balancing scheme (No LB) vs LDM vs SDM on the system for the random and round robin placement. Fig. 6 compares different performance metrics between these techniques in term of the maximum broker load, the *cov* value and the total number of subscriber migrations for the random placement. Since the load distribution in this scenario is less skewed, a few subscriber migrations happened (Fig. 6(3)) and the shuffle was not invoked. Similarly, Fig. 8 and Fig. 9 present the load distribution of the broker network following the NR placement policy with No LB vs DM and DM + GSH. In our experiment setup, NR placement turns out to produce an extreme load imbalance situation because of the non uniform distribution of subscribers geographically. This extreme skewed load distribution calls out the *shuffle* process. Fig. 10 and Fig. 11 respectively show the performance metric comparison for the NR placement under DM only vs combination of DM and GSH. We conclude that our load balancing techniques are able to fix the extreme cases of load imbalance as in the NR placement and we can also be able to improve the load distribution under even a slight imbalance as in the RND placement (reduce the *cov* value and the maximum broker load). However, the costs in terms of the number of subscriber migrations needed are not the same as shown in Fig. 12. The total number of subscriber migrations required for the NR placement are much higher compared with the RR and RND policies in both LDM and SDM techniques.

Subscription similarity exploitation: As we can see from the Fig. 6 and Fig. 10, the performances of SDM and LDM techniques are similar in these scenarios as we have already taken into account the subscription similarity among subscribers and between subscribers and brokers in the way we formulate the broker load.

Effect of the shuffle strategy: Fig. 11 shows that GSH at an early stage can help to fix poor subscribers-to-brokers assignments. The GSH scheme reduces the requirement for excessive migrations in the future as shown in Fig. 9 (3) when early GSH scheme is initiated compared with Fig. 8 (3) when only DM is applied. In our prototype experiment, the GSH scheme does not show much benefits compared to the DM strategy itself. The effect of the GSH technique will be further evaluated in the simulation experiment.

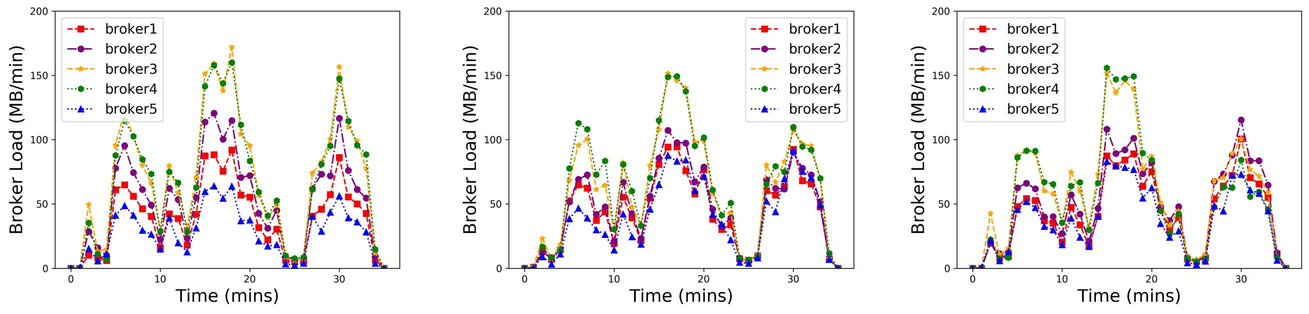

Figure 5: Broker load distribution in RND placement: (1) RND + No LB (2) RND + LDM (3) RND + SDM

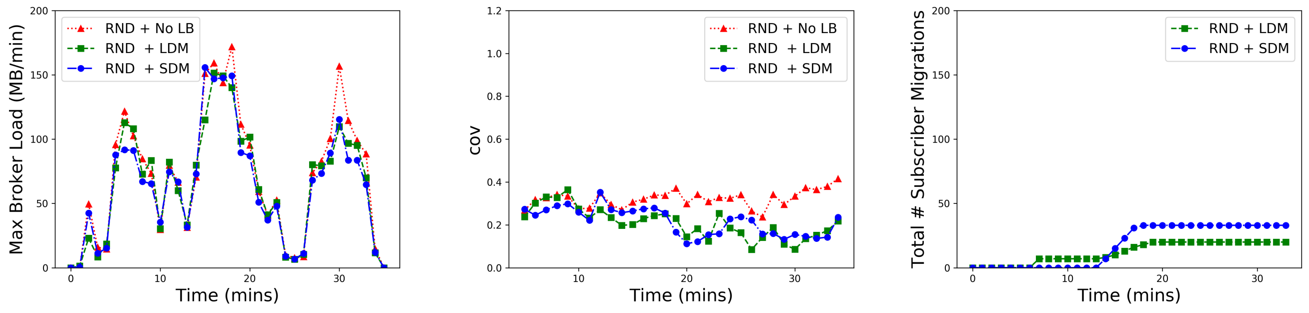

Figure 6: Performance metric comparison for RND placement: (1) max broker load (2) cov (3) total # of subscriber migration

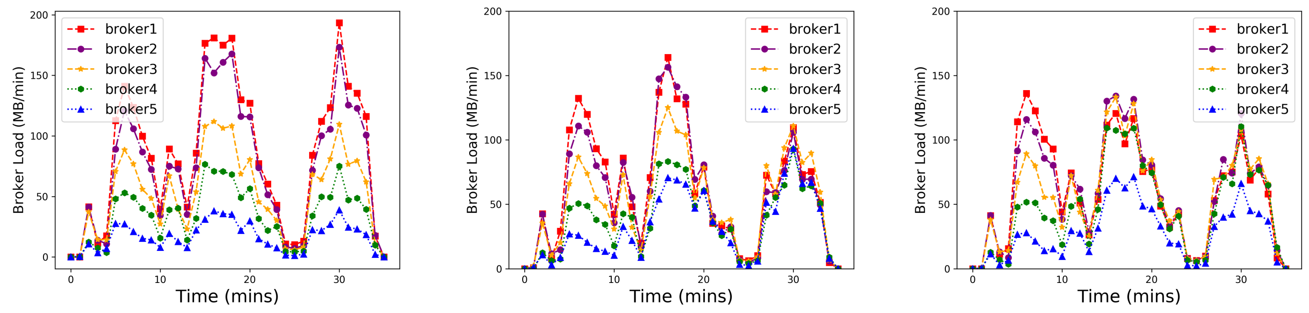

Figure 7: Broker load distribution in RR placement: (1) RR + No LB (2) RR + LDM (3) RR + SDM

6.2 Simulation Experiment

For the simulation, we develop a simulator written in Python3 that mimics the messaging level interactions among the BAD Pub/Sub components (data cluster, brokers, BCS and subscribers). We aim to scale up the experiments to further evaluate our load balancing schemes. We also analyze the effect of value selections of parameters α and β used in Algorithm 1 towards our LB techniques.

6.2.1 Simulation Setup. In our simulation setting, we model 10 brokers to support 10,000 subscribers. We create 10 channels having execution periods every 5, 10, 20, 30, 60 seconds and each channel having 100 different subscriptions. This setup creates a total of 1,000 BE subscriptions supported by the system. Each BE subscription has a data rate that follows a Normal distribution with a predefined mean and standard deviation values. Subscribers make a random number of subscriptions in the range of (10, 30) to those 10 defined channels with different parameters and create roughly 200,000 FE subscriptions in total. Each simulation run for half an hour where all subscribers login to the system at the beginning and make subscriptions in the first 8 minutes. During the course of execution, the data rates of some BE subscriptions from some channels may fluctuate. In detail, we select one-fourth of the subscriptions every minute to schedule them to increase the data rates within the next 3 minutes. Those subscriptions with increased data rates are then scheduled to decrease the data rates after 4 to 6 minutes. Our purpose is to produce a dynamic system load.

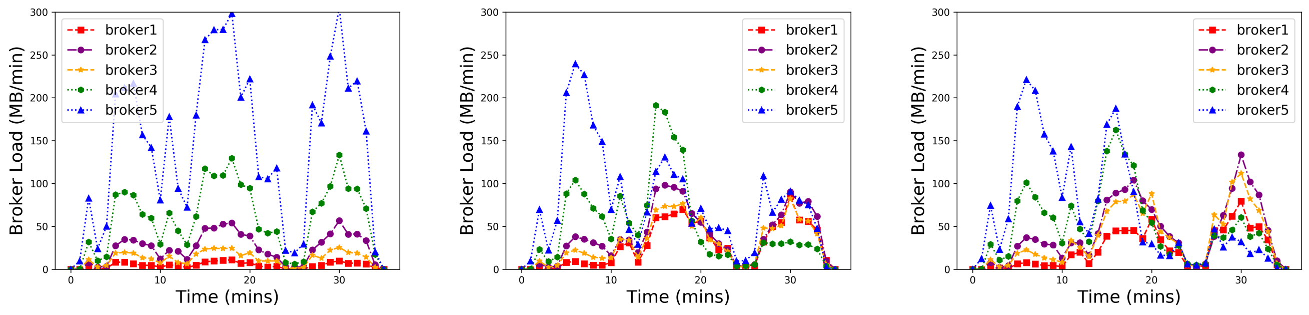

Figure 8: Broker load distribution in NR placement: (1) NR + No LB (2) NR + LDM (3) NR + SDM

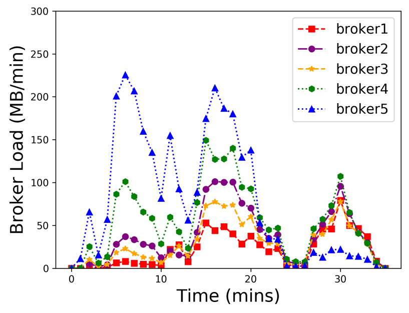

Figure 9: Broker load distribution in NR placement: (1) NR + LDM + GSH (2) NR + SDM + GSH

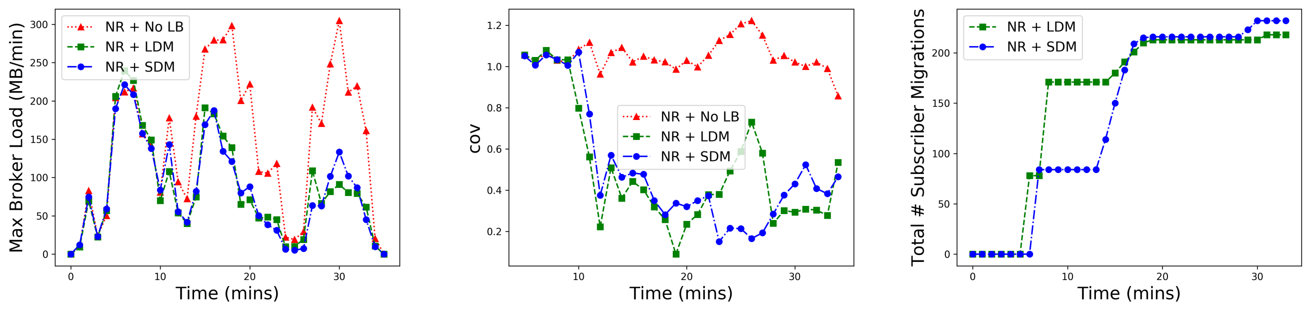

Figure 10: Performance metric comparison for NR placement with DM: (1) max broker load (2) cov (3) total # subscriber migrations

6.2.2 Simulation Results. We evaluate our proposed LDM and GSH techniques using the NR policy as the initial placement and the sensitivity of threshold values (α and β) we use towards our overall LB techniques.

LDM vs GSH: Fig. 13 (1) illustrates the client distribution among brokers and the broker load distribution when BCS assigns subscribers to brokers according to their geo-locations. The skewed distribution of subscribers geographically results in the unbalanced allocation of subscribers among brokers. There is a broker assigned with more than 2000 subscribers while other brokers may only need to support less than 500 subscribers. We can see the high variation of load between brokers in the NR setting (Fig. 13 (1)). We then apply respectively the LDM (α = 0.15 and β = 300 MB/sec) and GSH techniques as shown in Fig. 13 (2) and Fig. 13 (3). Both LDM and GSH techniques can fix the load imbalance eventually, but the GSH technique can fix the imbalance immediately after the process terminates and requires no further migration afterward while the LDM takes longer time to finally reach the balanced state.

Sensitivity of threshold value selection: Fig. 14 demonstrates the effect of value selection for the threshold parameters to the performance of our LB methods. As we can see from the Fig. 14 (1), when we set the α and β values are very small (α = .05 and β = 300

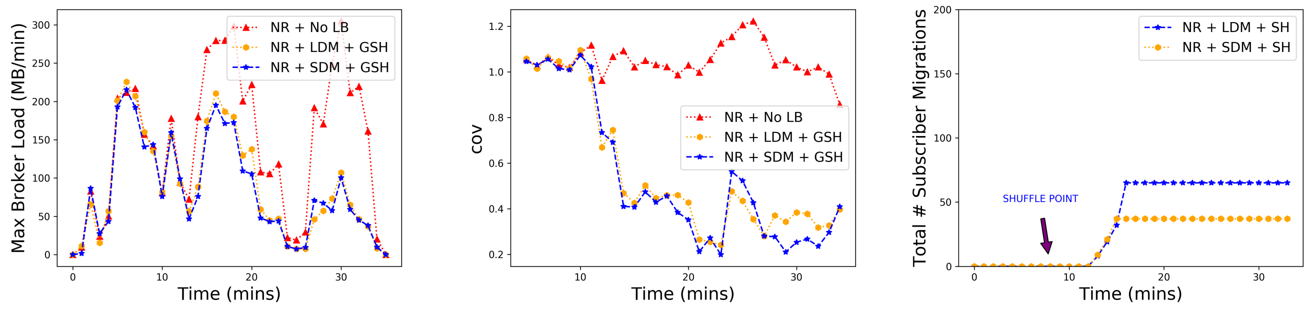

Figure 11: Performance metric comparison for NR placement with DM and GSH: (1) max broker load (2) cov (3) total # subscriber migrations

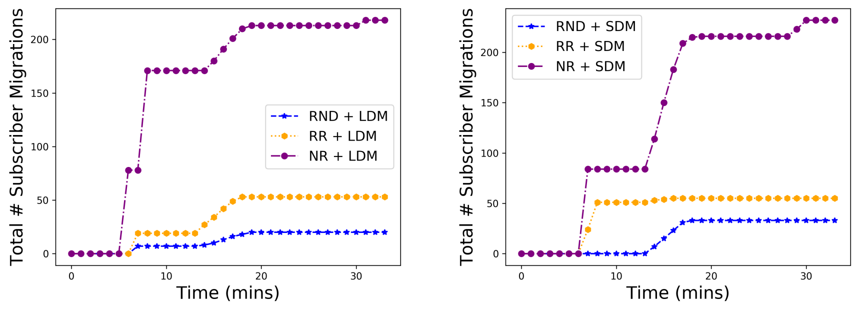

Figure 12: Total # subscriber migrations comparison for RND, RR and NR placements: (1) LDM (2) SDM

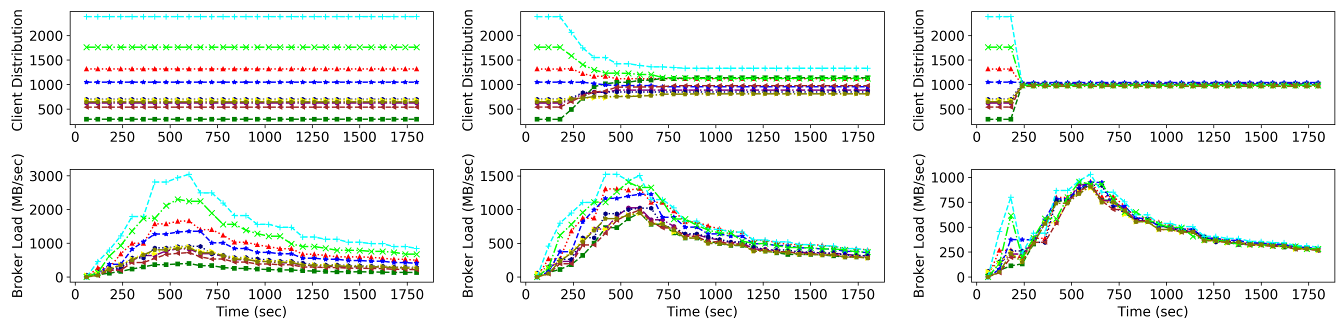

Figure 13: Broker load distribution: (1) NR placement + No LB (2) NR + LDM (3) NR + GSH

MB/sec in our simulation setting), the LDM scheme can achieve as good as the GSH scheme after a while. The smaller the values of α and β parameters, the more sensitive of the system towards the DM process as a slight load imbalance can invoke the DM process. This helps the system to achieve a better balanced state but with the cost of higher number of migrations as shown in Fig. 14 (2). Therefore, we set the value of α and β not too small in DM techniques to avoid redundant migrations due to small peaks or slight imbalances in the broker network. GSH scheme only comes into play when the system experiences an extreme skewed load distribution. The GSH scheme can quickly reconfigure the assignments of subscribers to brokers and achieve a better uniform load distribution for the system at the moment Fig. 13 (3).

7 CONCLUSION

In this paper, we propose and demonstrate the effectiveness of our multistage adaptive load balancing strategy which triggers different load balancing schemes at different stages of the system depending on the degree of skewed load distribution. Our load balancing framework composes of three main phases: *initial placement*, *dynamic migration* and *shuffle*. We did intensive evaluation of two load balancing schemes under both test-bed implementation and

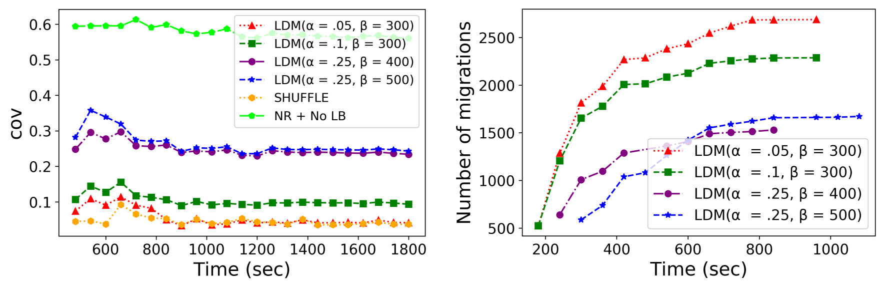

Figure 14: Evaluation of α and β values towards: (1) cov (2) number of migrations

simulation experiment in different placement scenarios. We exploit the subscription similarity feature to the extend of broker load formulation and in the SDM technique when choosing the destination broker for the migrated subscriber. For the future work, we aim to exploit subscription similarity more thoroughly in the sense that we may cluster subscribers and perform subscribers migration in groups based on the subscription similarity among subscribers and between subscribers and brokers. In the next step, we will setup our prototype implementation onto suitable cloud platform for experiments with real world workloads. We will further investigate into the failure detection and state management problems in the broker network as our next research topics.

REFERENCES

[1] Uddin, Md Yusuf Sarwar, and Nalini Venkatasubramanian. "Edge Caching for Enriched Notifications Delivery in Big Active Data." 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2018.
[2] Jacobs, Steven, et al. "A BAD demonstration: towards Big Active Data." Proceedings of the VLDB Endowment 10.12 (2017): 1941-1944.
[3] Jacobs, Steven. A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform. Diss. UC Riverside, 2018.
[4] Carey, Michael J., Steven Jacobs, and Vassilis J. Tsotras. "Breaking BAD: a data serving vision for big active data." Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. ACM, 2016.
[5] "Big Active Data" [Online]. Available: http://asterix.ics.uci.edu/bigactivedata/
[6] Dedousis, Dimitris, Nikos Zacheilas, and Vana Kalogeraki. "On the fly load balancing to address hot topics in topic-based pub/sub systems." 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2018.
[7] Zhao, Ye, Kyungbaek Kim, and Nalini Venkatasubramanian. "Dynatops: A dynamic topic-based publish/subscribe architecture." Proceedings of the 7th ACM international conference on Distributed event-based systems. ACM, 2013.
[8] Jafarpour, Hojjat, Sharad Mehrotra, and Nalini Venkatasubramanian. "Dynamic load balancing for cluster-based publish/subscribe system." Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on. IEEE, 2009.
[9] Yeung Cheung, Alex King, and Hans-Arno Jacobsen. "Dynamic load balancing in distributed content-based publish/subscribe." Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware. Springer-Verlag New York, Inc., 2006.
[10] Cheung, Alex King Yeung, and Hans-Arno Jacobsen. "Load balancing content-based publish/subscribe systems." ACM Transactions on Computer Systems (TOCS) 28.4 (2010): 9.
[11] Cheung, Alex King Yeung, and Hans-Arno Jacobsen. "Green resource allocation algorithms for publish/subscribe systems." Distributed Computing Systems (ICDCS), 2011 31st International Conference on. IEEE, 2011.
[12] Xia, Feng, et al. "Community-based event dissemination with optimal load balancing." IEEE Transactions on Computers 64.7 (2015): 1857-1869.
[13] Carey, Michael J., and Hongjun Lu. Load balancing in a locally distributed DB system. Vol. 15. No. 2. ACM, 1986.
[14] Eager, Derek L., Edward D. Lazowska, and John Zahorjan. "A comparison of receiver-initiated and sender-initiated adaptive load sharing." Performance evaluation 6.1 (1986): 53-68.
[15] Pai, Vivek S., et al. "Locality-aware request distribution in cluster-based network servers." ACM Sigplan Notices 33.11 (1998): 205-216.
[16] Huq, Sikder, et al. "Distributed Load Balancing in Key-Value Networked Caches." Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017.
[17] Ma, Ming, et al. "Joint Request Balancing and Content Aggregation in Crowdsourced CDN." Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017.
[18] Maguluri, Siva Theja, R. Srikant, and Lei Ying. "Stochastic models of load balancing and scheduling in cloud computing clusters." INFOCOM, 2012 Proceedings IEEE. IEEE, 2012.
[19] Rao, Ananth, et al. "Load balancing in structured P2P systems." International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2003.
[20] Gupta, Gaurav, and Mohamed F. Younis. "Load-balanced clustering of wireless sensor networks." ICC. Vol. 3. 2003.
[21] Low, Chor Ping, et al. "Efficient load-balanced clustering algorithms for wireless sensor networks." Computer Communications 31.4 (2008): 750-759.
[22] Zacheilas, Nikos, et al. "Dynamic load balancing techniques for distributed complex event processing systems." Distributed Applications and Interoperable Systems. Springer, Cham, 2016.
[23] Gupta, Abhishek, et al. "Meghdoot: content-based publish/subscribe over P2P networks." ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2004.
[24] Zhu, Yingwu, and Yiming Hu. "Efficient, proximity-aware load balancing for DHT-based P2P systems." IEEE Transactions on parallel and distributed systems 16.4 (2005): 349-361.
[25] Godfrey, Brighten, et al. "Load balancing in dynamic structured P2P systems." INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies. Vol. 4. IEEE, 2004.
[26] Salehi, Pooya, Kaiwen Zhang, and Hans-Arno Jacobsen. "PopSub: Improving resource utilization in distributed content-based publish/subscribe systems." Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. ACM, 2017.
[27] Chen, Chen, Hans-Arno Jacobsen, and Roman Vitenberg. "Algorithms based on divide and conquer for topic-based publish/subscribe overlay design." IEEE/ACM Transactions on Networking 24.1 (2016): 422-436.
[28] Turau, Volker, and Gerry Siegemund. "Scalable routing for topic-based publish/subscribe systems under fluctuations." Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017.
[29] Ji, Shuping, et al. "MERC: Match at Edge and Route intra–Cluster for Content-based Publish/Subscribe Systems." Proceedings of the 16th Annual Middleware Conference. ACM, 2015.
[30] Chen, Chen, Yoav Tock, and Sarunas Girdzijauskas. "BeaConvey: Co-Design of Overlay and Routing for Topic-basedPublish/Subscribe on Small-World Networks." 12th International Conference on Distributed and Event-Based Systems (DEBS 2018). 2018.
[31] Gascon-Samson, Julien, et al. "Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud." Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on. IEEE, 2015.
[32] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM Computer Communication Review 31.4 (2001): 149-160.
[33] Rowstron, Antony, and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems." IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2001.
[34] Coffman, Jr, Edward G., Michael R. Garey, and David S. Johnson. "An application of bin-packing to multiprocessor scheduling." SIAM Journal on Computing 7.1 (1978): 1-17.
[35] "Apache Asterix DB" [Online]. Available: https://asterixdb.apache.org
[36] . ONE Simulartor https://akeranen.github.io/the-one/