A Scalable and Reliable Matching Service for Content-based Publish/Subscribe Systems

Xingkong Ma, Student Member, IEEE, Yijie Wang, Member, IEEE, and Xiaqiang Pei

Abstract—Characterized by the increasing arrival rate of live content, the emergency applications pose a great challenge: how to disseminate large-scale live content to interested users in a scalable and reliable manner. The publish/subscribe (pub/sub) model is widely used for data dissemination because of its capacity of seamlessly expanding the system to massive size. However, most event matching services of existing pub/sub systems either lead to low matching throughput when matching a large number of skewed subscriptions, or interrupt dissemination when a large number of servers fail. The cloud computing provides great opportunities for the requirements of complex computing and reliable communication. In this paper, we propose SREM, a scalable and reliable event matching service for content-based pub/sub systems in cloud computing environment. To achieve low routing latency and reliable links among servers, we propose a distributed overlay SkipCloud to organize servers of SREM. Through a hybrid space partitioning technique HPartition, large-scale skewed subscriptions are mapped into multiple subspaces, which ensures high matching throughput and provides multiple candidate servers for each event. Moreover, a series of dynamics maintenance mechanisms are extensively studied. To evaluate the performance of SREM, 64 servers are deployed and millions of live content items are tested in a CloudStack testbed. Under various parameter settings, the experimental results demonstrate that the traffic overhead of routing events in SkipCloud is at least 60% smaller than in Chord overlay, the matching rate in SREM is at least 3.7 times and at most 40.4 times larger than the single-dimensional partitioning technique of BlueDove. Besides, SREM ensures the event loss rate to drop back to 0 in tens of seconds even if a large number of servers fail simultaneously.

Index Terms—Publish/Subscribe, Event Matching, Overlay Construction, Content Space Partitioning, Cloud Computing

1 INTRODUCTION

Because of the importance in helping users to make real-time decisions, data dissemination has become dramatically significant in many large-scale emergency applications, such as earthquake monitoring, disaster weather warning, and status update in social networks. Recently, data dissemination in these emergency applications presents a number of fresh trends. One is the rapid growth of live content. For instance, Facebook users publish over 600,000 pieces of content and Twitter users send over 100,000 tweets on average per minute [1]. The other is the highly dynamic network environment. For instance, the measurement studies indicates that most users’ sessions in social networks only last several minutes [2]. In emergency scenarios, the sudden disasters like earthquake or bad weather may lead to the failure of a large number of users instantaneously.

These characteristics require the data dissemination system to be scalable and reliable. Firstly, the system must be scalable to support the large amount of live content. The key is to offer a scalable event matching service to filter out irrelevant users. Otherwise, the content may have to traverse a large number of uninterested users before they reach interested users. Secondly, with the dynamic network environment, it’s quite necessary to provide reliable schemes to keep continuous data dissemination capacity. Otherwise, the system interruption may cause the live content becomes obsolete content.

Driven by these requirements, publish/subscribe (pub/sub) pattern is widely used to disseminate data due to its flexibility, scalability, and efficient support of complex event processing. In pub/sub systems (pub/subs), a receiver (subscriber) registers its interest in the form of a subscription. Events are published by senders to the pub/sub system. The system matches events against subscriptions and disseminates them to interested subscribers.

In traditional data dissemination applications, the live content are generated by publishers at a low speed, which makes many pub/subs adopt the multi-hop routing techniques to disseminate events. A large body of broker-based pub/subs forward events and subscriptions through organizing nodes into diverse distributed overlays, such as tree-based design [3]–[6], cluster-based design [7], [8] and DHT-based design [9]–[11]. However, the multi-hop routing techniques in these broker-based systems lead to a low matching throughput, which is inadequate to apply to current high arrival rate of live content.

Recently, cloud computing provides great opportunities for the applications of complex computing and high speed communication [12], where the servers are connected by high speed networks, and have pow-
erful computing and storage capacities. A number of pub/sub services based on the cloud computing environment have been proposed, such as Move [13], BlueDove [14] and SEMAS [15]. However, most of them cannot completely meet the requirements of both scalability and reliability when matching large-scale live content under highly dynamic environments. This mainly stems from the following facts: 1) Most of them are inappropriate to the matching of live content with high data dimensionality due to the limitation of their subscription space partitioning techniques, which bring either low matching throughput or high memory overhead. 2) These systems adopt the one-hop lookup technique [16] among servers to reduce routing latency. In spite of its high efficiency, it requires each dispatching server to have the same view of matching servers. Otherwise, the subscriptions or events may be assigned to the wrong matching servers, which brings the availability problem in the face of current joining or crash of matching servers. A number of schemes can be used to keep the consistent view, like periodically sending heartbeat messages to dispatching servers or exchanging messages among matching servers. However, these extra schemes may bring a large traffic overhead or the interruption of event matching service.

Motivated by these factors, we propose a scalable and reliable matching service for content-based pub/sub service in cloud computing environments, called SREM. Specifically, we mainly focus on two problems: one is how to organize servers in the cloud computing environment to achieve scalable and reliable routing. The other is how to manage subscriptions and events to achieve parallel matching among these servers. Generally speaking, we provide the following contributions:

- We propose a distributed overlay protocol, called SkipCloud, to organize servers in the cloud computing environment. SkipCloud enables subscriptions and events to be forwarded among brokers in a scalable and reliable manner. Also it is easy to implement and maintain.
- To achieve scalable and reliable event matching among multiple servers, we propose a hybrid multi-dimensional space partitioning technique, called HPartition. It allows similar subscriptions to be divided into the same server and provides multiple candidate matching servers for each event. Moreover, it adaptively alleviates hot spots and keeps workload balance among all servers.
- We implement extensive experiments based on a CloudStack testbed to verify the performance of SREM under various parameter settings.

The rest of this paper is organized as follows. Section 2 introduces content-based data model and an essential framework of SREM. Section 3 presents the SkipCloud overlay in detail. Section 4 describes HPartition in detail. Section 5 discusses the dynamics maintenance mechanisms of SREM. We evaluate the performance of SREM in Section 6. In Section 7, we review the related work on the matching of existing content-based pub/subs. Finally, we conclude the paper and outline our future work in Section 8.

2 DESIGN OF SREM

2.1 Content-Based Data Model

SREM uses a multi-dimensional content-based data model. Consider our data model consists of $k$ dimensions $A_1, A_2, \ldots, A_k$. Let $R_i$ be the ordered set of all possible values of $A_i$. So, $\Omega = R_1 \times R_2 \times \cdots \times R_k$ is the entire content space. A subscription is a conjunction of predicates over one or more dimensions. Each predicate $P_i$ specifies a continuous range for a dimension $A_i$, and it can be described by the tuple $(A_i, v_i, O_i)$, where $v_i \in R_i$ and $O_i$ represents a relational operator ($<, \leq, \neq, \geq, >$, etc). The general form of a subscription is $S = \wedge_{i=1}^{k} P_i$. An event is a point within the content space $\Omega$. It can be represented as $k$ dimension-value pairs, i.e., $e = \wedge_{j=1}^{k} (A_j, v_j)$. For each pair $(A_j, v_j)$, we say it satisfies a predicate $(A_i, v_i, O_i)$ if $A_j = A_i$ and $v_j O_i v_i$. By this definition we say an event $e$ matches $S$ if each predicate of $S$ satisfies some pairs of $e$.

2.2 Overview of SREM

![System Framework](image)

To support large-scale users, we consider a cloud computing environment with a set of geographically distributed datacenters through the Internet. Each datacenter contains a large number of servers (brokers), which are managed by a datacenter management service such as Amazon EC2 or OpenStack.

We illustrate a simple overview of SREM in Figure 1. All brokers in SREM as the front-end are exposed to the Internet, and any subscriber and publisher can connect to them directly. To achieve reliable connectivity and low routing latency, these brokers are connected through an distributed overlay, called SkipCloud. The entire content space is partitioned into disjoint subspaces, each of which is managed by a number of brokers. Subscriptions and events are dispatched to the subspaces that are overlapping.
with them through SkipCloud. Thus, subscriptions and events falling into the same subspace are matched on the same broker. After the matching process completes, events are broadcasted to the corresponding interested subscribers. As shown in Figure 1, the subscriptions generated by subscribers $S_1$ and $S_2$ are dispatched to broker $B_2$ and $B_5$, respectively. Upon receiving events from publishers, $B_2$ and $B_5$ will send matched events to $S_1$ and $S_2$, respectively.

One may argue that different datacenters are responsible for some subset of the subscriptions according to the geographical location, such that we do not really need much collaboration among the servers [3], [4]. In this case, since the pub/sub system needs to find all the matched subscribers, it requires each event to be matched in all datacenters, which leads to large traffic overhead with the increasing number of datacenters and the increasing arrival rate of live content. Besides, it’s hard to achieve workload balance among the servers of all datacenters due to the various skewed distributions of users’ interests. Another question is that why we need a distributed overlay like SkipCloud to ensure reliable logical connectivity in datacenter environment where servers are more stable than the peers in P2P networks. This is because as the number of servers increases in datacenters, the node failure becomes normal, but not rare exception [17]. The node failure may lead to unreliable and inefficient routing among servers. To this end, we try to organize servers into SkipCloud to reduce the routing latency in a scalable and reliable manner.

Such a framework offers a number of advantages for real-time and reliable data dissemination. First, it allows the system to timely group similar subscriptions into the same broker due to the high bandwidth among brokers in the cloud computing environment, such that the local searching time can be greatly reduced. This is critical to reach high matching throughput. Second, since each subspace is managed by multiple brokers, this framework is fault-tolerant even if a large number of brokers crash instantaneously. Third, because the datacenter management service provides scalable and elastic servers, the system can be easily expanded to Internet-scale.

3 SkipCloud

3.1 Topology Construction

Generally speaking, SkipCloud organizes all brokers into levels of clusters. As shown in Figure 2, the clusters at each level of SkipCloud can be treated as a partition of the whole broker set. Table 1 shows key notations used in this section.

![Fig. 2: An example of SkipCloud with 8 brokers and 3 levels. Each broker is identified by a binary string.](image)

At the top level, brokers are organized into multiple clusters whose topologies are complete graphs. Each cluster at this level is called top cluster. It contains a leader broker which generates a unique $b$-ary identifier with length of $m$ using a hash function (e.g. MD-5). This identifier is called ClusterID. Correspondingly, each broker’s identifier is a unique string with length of $m+1$ and shares common prefix of length $m$ with its ClusterID. At this level, brokers in the same cluster are responsible for the same content subspaces, which provides multiple matching candidates for each event. Since brokers in the same top cluster generate frequent communication among themselves, such as updating subscriptions and dispatching events, they are organized into a complete graph to reach each other in one hop.

After the top clusters have been well organized, the clusters at the rest levels can be generated level by level. Specifically, each broker decides to join which cluster at every level. The brokers whose identifiers share the common prefix with length $i$ would join the same cluster at level $i$, and the common prefix is referred to as the ClusterID at level $i$. That is, clusters at level $i+1$ can be regarded as a $b$-partition of the clusters at level $i$. Thus, the number of clusters reduces linearly with the decreasing of levels. Let $\varepsilon$ be the empty identifier. All brokers at level 0 join one single cluster, called global cluster. Therefore, there are $b^i$ clusters at level $i$. Figure 2 shows an example of how SkipCloud organizes 8 brokers into 3 levels of clusters by binary identifiers.

### Algorithm 1: Neighbor List Maintenance

```
Input : views[i]: the neighbor lists.
       m: the total number of levels in SkipCloud.
cycle: current cycle.

1. $j = cycle \% (m + 1)$;
2. for each $i$ in $[0,m-1]$ do
   3. update views[i] by the peer sampling service based on Cyclon.
4. for each $i$ in $[0,m-1]$ do
   5. if views[i] contains empty slots then
      6. fill these empty slots with other levels’ items who share common prefix of length $i$ with the ClusterID of views[i].
```
To organize clusters of the non-top levels, we employ a light-weighted peer sampling protocol based on Cyclon [18], which provides robust connectivity of each cluster. Suppose there are $m$ levels in SkipCloud. Specifically, each cluster runs Cyclon to keep reliable connectivity among brokers in the cluster. Since each broker falls into a cluster at each level, it maintains $m$ neighbor lists. For a neighbor list at the cluster of level $i$, it samples equal number of neighbors from their corresponding children clusters at level $i + 1$. This ensures that the routing from the bottom level can always find a broker pointing to a higher level. Because brokers maintain levels of neighbor lists, they update the neighbor list of one level and the subsequent ones in a ring to reduce the traffic cost. The pseudo-code of view maintenance algorithm is show in Algorithm 1. This topology of the multi-level neighbor lists is similar to Tapestry [19]. Compared with Tapestry, SkipCloud uses multiple brokers in top clusters as targets to ensure reliable routing.

### 3.2 Prefix Routing

Prefix routing in SkipCloud is mainly used to efficiently route subscriptions and events to the top clusters. Note that the cluster identifiers at level $i + 1$ are generated by appending one $b$-ary to the corresponding clusters at level $i$. The relation of identifiers between clusters is the foundation of routing to target clusters. Briefly, when receiving a routing request to a specific cluster, a broker examines its neighbor lists of all levels and chooses the neighbor which shares the longest common prefix with the target ClusterID as the next hop. The routing operation repeats until a broker can not find a neighbor whose identifier is more closer than itself. Algorithm 2 describes the prefix routing algorithm in pseudo-code.

#### Algorithm 2: Prefix Routing

```
1  $l = \text{commonPrefixLength}(self.ID, event.ClusterID);$
2  $\text{process}(event);$
3  $\text{else}$
4  $\text{destB} \leftarrow \text{the broker whose identifier matches event.ClusterID with}$
5  $\text{the longest common prefix from self.views}.$
6  $l_{\text{max}} = \text{commonPrefixLength(destB.identifier, event.ClusterID)};$
7  if ($l_{\text{max}} \leq l$)
8      $\text{destB} \leftarrow \text{the broker whose identifier is closest to}$
9      $\text{event.ClusterID from view[i].}$
10     $\text{if (destB and myself is in the same cluster of level l) then}$
11     \hspace{1em} $\text{process(event);}$
12  $\text{forwardTo(destB, event);}$
```

Since a neighbor list in the cluster at level $i$ is a uniform sampling from the corresponding children clusters at level $i + 1$, each broker can find a neighbor whose identifier matches at least one more longer common prefix with the target identifier before reaching the target cluster. Therefore, the prefix routing in Algorithm 2 guarantees that any top cluster will be reached in at most $\log_b N_c$ hops, where $N_c$ is the the number of top clusters. Assume the average degree of brokers in each cluster is $D_c$. Thus, each broker only needs to keep $D_c \ast \log_b N_c$ neighbors.

### 4 HPartition

In order to take advantage of multiple distributed brokers, SREM divides the entire content space among the top clusters of SkipCloud, so that each top cluster only handles a subset of the entire space and searches a small number of candidate subscriptions. SREM employs a hybrid multi-dimensional space partitioning technique, called HPartition, to achieve scalable and reliable event matching. Generally speaking, HPartition divides the entire content space into disjoint subspaces (Section 4.1). Subscriptions and events with overlapping subspaces are dispatched and matched on the same top cluster of SkipCloud (Section 4.2 and 4.3). To keep workload balance among servers, HPartition divides the hot spots into multiple cold spots in an adaptive manner (Section 4.4). Table 2 shows key notations used in this section.

<table>
<thead>
<tr>
<th>Table 2: Notations in HPartition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Omega$</td>
</tr>
<tr>
<td>$k$</td>
</tr>
<tr>
<td>$A_i$</td>
</tr>
<tr>
<td>$R_i$</td>
</tr>
<tr>
<td>$P_i$</td>
</tr>
<tr>
<td>$N_{seg}$</td>
</tr>
<tr>
<td>$N_{seg}$</td>
</tr>
<tr>
<td>$\alpha$</td>
</tr>
<tr>
<td>$N_{sub}$</td>
</tr>
<tr>
<td>$\Omega_{x_1 \ldots x_k}$</td>
</tr>
</tbody>
</table>

Fig. 3: Comparison among three different logical space construction techniques.

![Fig. 3: Comparison among three different logical space construction techniques.](image)
The basic idea of SPartition like BlueDove [14] is to treat each dimension as a separated space, as show in Figure 3 (a). Specifically, the range of each dimension is divided into $N_{seg}$ segments, each of which is regarded as a separated subspace. Thus, the entire space $\Omega$ is divided into $k \times N_{seg}$ subspaces. Subscriptions and events falling into the same subspace are matched against each other. Due to the coarse-grained partitioning of SPartition, each subscription falls into a small number of subspaces, which brings multiple candidate brokers for each event and low memory overhead. On the other hand, SPartition may fail into a small number of subspaces, which brings a higher memory cost. On the other hand, APartition only has one candidate broker. Compared with SPartition, each subscription in APartition falls into more subspaces, which leads to a higher memory cost.

On the other hand, the idea of APartition like SEMAS [15] is to treat the combination of all dimensions as a separate space, as shown in Figure 3 (b). Formally, the whole space $\Omega$ is partitioned into $(N_{seg})^k$ subspaces. Compared with SPartition, APartition leads to a smaller number of hot spots, since a hot subspace would be formed only if its all segments are subscribed by a large number of subscriptions. On the other hand, each event in APartition only has one candidate broker. Compared with SPartition, each subscription in APartition falls into more subspaces, which leads to a higher memory cost.

Inspired by both SPartition and APartition, HPartition provides a flexible manner to construct logical space. Its basic idea is to divide all dimensions into a number of groups and treat each group as a separate space, as shown in Figure 3 (c). Formally, $k$ dimensions in HPartition are classified into $t$ groups, each of which contains $k_i$ dimensions, where $\sum_{i=1}^{t} k_i = k$. That is, $\Omega$ is split into $t$ separated spaces $\Omega_i, i \in [1, t]$. HPartition adopts APartition to divide each $\Omega_i$. Thus, $\Omega$ is divided into $\sum_{i=1}^{t} (N_{seg})^{k_i}$ subspaces. For each space $\Omega_i$, let $G_{x_1 \cdots x_k}, x_i \in [0, N_{seg}]$ be its subspace, where $x_1 \cdots x_k$ is the concatenation of SegIDs of the subspace, called SubspaceID. For the dimensions that a subspace does not contain, the corresponding positions of the SubspaceID are set to 0.

Each subspace of HPartition is managed by a top cluster of SkipCloud. To dispatch each subspace $G_{x_1 \cdots x_k}$ to its corresponding top cluster of SkipCloud, HPartition uses a hash function like MurMurHash [20] to map each SubspaceID $x_1 \cdots x_k$ to a $b$-ary identifier with length of $m$, represented by $H(x_1 \cdots x_k)$. According to the prefix routing in Algorithm 2, each subspace will be forwarded to the top cluster whose ClusterID is nearest to $H(x_1 \cdots x_k)$. When a broker in the top cluster receives the subscription $\Omega_i$, it broadcasts $\Omega_i$ to other brokers in the same cluster, such that the top cluster provides reliable event matching and balanced workloads among its brokers.

4.2 Subscription Installation

A user can specify a subscription by defining the ranges over one or more dimensions. We treat $S = \bigwedge_{i=1}^{k} P_i$ as a general form of subscriptions, where $P_i$ is a predicate of the dimension $A_i$. If $S$ does not contain a dimension $A_i$, the corresponding $P_i$ is the entire range of $A_i$. When receiving a subscription $S$, the broker first obtains all subspaces $G_{x_1 \cdots x_k}$ which are overlapping with $S$. Based on the logical space construction of HPartition, all dimensions are classified into $t$ individual spaces $\Omega_j, j \in [1, t]$. For each $G_{x_1 \cdots x_k}$ of $\Omega_i$, it satisfies

$$x_i = \begin{cases} s, & \text{if } A_i \neq R_i, R_i \cap P_i \neq \emptyset \\ 0, & \text{if } A_i = R_i \end{cases}$$

(1)

As shown in in Figure 4, $\Omega$ consists of 2 groups, and each range is divided into 2 segments. For a subscription $S_1 = (20 \leq A_1 \leq 80) \land (80 \leq A_2 \leq 100) \land (105 \leq A_3 \leq 170) \land (92 \leq A_4 \leq 115)$, its matched subspaces are $G_{1200}, G_{1100}$ and $G_{0022}$.

![Fig. 4: An example of subscription assignment.](image)

For the subscription $S_1$, its every matched subspace $G_{x_1 \cdots x_k}$ is hashed to a $b$-ary identifier with length of $m$, represented by $H(x_1 \cdots x_k)$. Thus, $S_1$ is forwarded to the top cluster whose ClusterID is nearest to $H(x_1 \cdots x_k)$. When a broker in the top cluster receives the subscription $S_1$, it broadcasts $S_1$ to other brokers in the same cluster, such that the top cluster provides reliable event matching and balanced workloads among its brokers.

4.3 Event Assignment and Forwarding Strategy

Upon receiving an event, the broker forwards this event to its corresponding subspaces. Specifically, considering an event $e = \bigwedge_{i=1}^{k} (A_i, v_i)$. Each subspace $G_{x_1 \cdots x_k}$ that $e$ falls into should satisfy

$$x_i \in \{0, s\}, v_i \in R^e_i$$

(2)

For instance, let $e$ be $(A_1, 30) \land (A_2, 100) \land (A_3, 80) \land (A_4, 80)$. Based on the logical partitioning in Figure 4, the matched subspaces of $e$ are $G_{1200}, G_{0011}$.

Recall that HPartition divides the whole space $\Omega$ into $t$ separated spaces $\Omega_i$, which indicates there are $t$ candidate subspaces $\{G_i, 1 \leq i \leq t\}$ for each event. Because of the different workloads, the strategy
of how to select one candidate subspace for each event may greatly affect the matching rate. For an event \( e \), suppose each candidate subspace \( G_i \) contains \( N_i \) subscriptions. The most intuitive approach is the least first strategy, which chooses the subspace with the least number of candidate subscriptions. Correspondingly, its average searching subscription amount is \( N_{\text{least}} = \min_{i \in [1, t]} N_i \). However, when a large number of skewed events fall into the same subspace, this approach may lead to unbalanced workloads among brokers. Another is the random strategy, which selects each candidate subspace with equal probability. It ensures balanced workloads among brokers. However, the average searching subscription amount increases to \( N_{\text{avg}} = \frac{1}{t} \sum_{i=1}^{t} N_i \). In HPartition, we adopt a probability-based forwarding strategy to dispatch events. Formally, the probability of forwarding \( e \) to the space \( \{\Omega_i\} \) is \( p_i = (1 - N_i / \sum_{i=1}^{t} N_i) / (t - 1) \). Thus, the average searching subscription amount is \( N_{\text{prob}} = (N - \sum_{i=1}^{t} N_i^2) / (t - 1) \). Note that when every subspace has the same size, \( N_{\text{prob}} \) reaches its maximal value, i.e., \( N_{\text{prob}} \leq N_{\text{avg}} \). Therefore, this approach has better workload balance than the least first strategy and less searching latency than the random strategy.

### 4.4 Hot Spots Alleviation

In real applications, a large number of subscriptions fall into a few number of subspaces which are called hot spots. The matching in hot spots leads to a high searching latency. To alleviate the hot spots, an obvious approach is to add brokers to the top clusters containing hot spots. However, this approach raises the total price of the service and wastes resources of the clusters only containing cold spots.

To utilize distributed multiple clusters, a better solution is to balance the workloads among clusters through partitioning and migrating hot spots. The gain of the partitioning technique is greatly affected by the distribution of subscriptions of the hot spot. To this end, HPartition divides each hot spot into a number of cold spots through two partitioning techniques: hierarchical subspace partitioning and subscription set partitioning. The first aims to partition the hot spots where the subscriptions are diffused among the whole space, and the second aims to partition the hot spots where the subscriptions fall into a narrow space.

#### 4.4.1 Hierarchical Subspace Partitioning

To alleviate the hot spots whose subscriptions diffuse in its whole space, we propose a hierarchical subspace partitioning technique, called HSPartition. Its basic idea is to divide the space of a hot spot into a number of smaller subspaces and reassign its subscriptions into these new generated subspaces.

Step one, each hot spot is divided along with its ranges of dimensions. Assume \( \Omega_i \) contains \( k_i \) dimensions, and one of its hot spots is \( G_{x_1 \cdots x_k} \). For each

\[ x_i \neq 0, \text{ the corresponding range } R_i^{x_i} \text{ is divided into } N_{seg}^{x_i} \text{ segments, each of which is denoted by } R_{x_i|y_i}(1 \leq x_i \leq N_{seg}^{x_i}, 1 \leq y_i \leq N_{seg}^{x_i}). \text{ Therefore, } G_{x_1 \cdots x_k} \text{ is divided into } (N_{seg}^{x_1})^{k} \text{ subspaces, each of which is represented by } G_{x_1 \cdots x_k|y_1 \cdots y_k}. \]

Step two, subscriptions in \( G_{x_1 \cdots x_k} \) are reassigned to these new subspaces according to the subscription installation in Section 4.2. Because each subspace occupies a small piece of the space of \( G_{x_1 \cdots x_k} \), it means a subscription \( S \) of \( G_{x_1 \cdots x_k} \) may fall into a part of these subspaces, which will reduce the number of subscriptions that each event matches against on the hot spot. HSPartition enables the hot spots to be divided into much smaller subspaces iteratively. As shown in the left top corner Figure 5, the hot spot \( G_{102} \) is divided into 4 smaller subspaces by HSPartition, and the maximum number of subscriptions in this space decreases from 9 to 4.

To avoid concurrency partitioning one hot spot by multiple brokers in the same top cluster, the leader broker of each top cluster is responsible for periodically checking and partitioning hot spots. When a broker receives a subscription, it hierarchically divides the space of the subscription until each subspace can not be divided further. Therefore, the leader broker that is in charge of dividing a hot spot \( G_{x_1 \cdots x_k} \) disseminates an update message including a three-tuple \( \{\text{“Subspace Partition”, } x_1 \cdots x_k, N_s’\} \) to all the other brokers. Because the global cluster at level 0 of SkipCloud organizes all brokers into a random graph, we can utilize a gossip-based multicast method [21] to disseminate the update message to other brokers in \( O(\log N_b) \) hops, where \( N_b \) is the total broker size.

#### 4.4.2 Subscription Set Partitioning

To alleviate the hot spots whose subscriptions fall into a narrow space, we propose a subscription set partitioning, called SSPartition. Its basic idea is to divide the subscriptions set of a hot spot into a number of subsets and scatter these subsets to multiple top clusters. Assume \( G_{x_1 \cdots x_k} \) is a hot spot and \( N_b \) is the size of each subscription subset.

Step one, divide the subscription set of \( G_{x_1 \cdots x_k} \) into \( n \) subsets, where \( n = |G_{x_1 \cdots x_k}|/N_b \). Each subset
is assigned a new SubspaceID $x_1 \cdots x_k - i$, $(1 \leq i \leq |G_{x_1 \cdots x_k}|/N_0)$, and the subscription with identifier SubID would be dispatched to the subset $G_{x_1 \cdots x_k - i}$ if $H(SubID) \% n = i$.

Step two, each subset $G_{x_1 \cdots x_k - i}$ is forwarded to its corresponding top cluster with ClusterID $H(x_1 \cdots x_k - i)$. Since these non-overlapping subsets are scattered to multiple top clusters, it allows events falling into the hot spots to be matched in parallel multiple brokers, which brings much lower matching latency. Similar to the process of HSPartition, the leader broker who is responsible for dividing a hot spot $G_{x_1 \cdots x_k}$ disseminates a three-tuple \{"Subscription Partition", $x_1 \cdots x_k$, $n$\} to all the other brokers to support further partitioning operations. As shown in the right bottom corner of Figure 5, the hot spot $G_{2100}$ into 3 smaller subsets $G_{2100-1}$, $G_{2100-2}$ and $G_{2100-3}$ by SSPartition. Correspondingly, the maximal number of subscriptions of the subspace decreases from 11 to 4.

4.4.3 Adaptive Selection Algorithm

Because of diverse distributions of subscriptions, both HSPartition and SSPartition can not replace with each other. On one hand, HSPartition is attractive to divide the hot spots whose subscriptions are uniform scattered regions. However, it’s inappropriate to divide the hot spots whose subscriptions all appear at the same exact point. On the other hand, SSPartition allows to divide any kind of hot spots into multiple subsets even if all subscriptions falls into the same single point. However, compared with HSPartition, it has to dispatch an event to multiple subspaces, which brings a higher traffic overhead.

To achieve balanced workloads among brokers, we propose an adaptive selection algorithm to select either HSPartition or SSPartition to alleviate hot spots. The selection is based on the similarity of subscriptions in the same hot spot. Specifically, assume $G_{x_1 \cdots x_k}$ is the subspace with maximal size of subscriptions in HSPartition. $\alpha$ is a threshold value, which represents the similarity degree of subscription- space in a hot spot. We choose HSPartition as the partitioning algorithm if $|G_{x_1 \cdots x_k|}/|G_{x_1 \cdots x_k|} < \alpha$. Otherwise, we choose SSPartition. Through combining both partitioning techniques, this selection algorithm can alleviate hot spots in an adaptive manner.

4.5 Performance Analysis

4.5.1 The Average Searching Size

Since each event is matched in one of its candidate subspaces, the average number of subscriptions of all subspaces, called the average searching size, is critical to reduce the matching latency. We give the formal analysis as follows.

Theorem 1: Suppose the percentage of each predicate’s range of each subscription is $\lambda$, then the average searching size $N_{\text{prob}}$ in HPartition is not more than $N_{\text{sub}} \frac{t}{t} \sum_{i=1}^{N_{\text{seg}}} \lambda^k$, if $N_{\text{seg}} \to \infty$, where $t$ is the number of groups in the entire space, $k_i$ is the number of dimensions in each group, and $N_{\text{sub}}$ is the number of subscriptions.

Proof: For each dimension $A_i$, $i \in [1, k]$, the range of $A_i$ is $R_i$, the length of corresponding predicate $P_i$ is $\lambda \| R_i \|$. Since each dimension is divided into $N_{\text{seg}}$ segments, the length of each segment is $\| R_i \|/N_{\text{seg}}$. Thus, the expected number of segments that $P_i$ falls into is $\lceil \lambda N_{\text{seg}} \rceil$. For a space $\Omega$, if it contains $(N_{\text{seg}})^k$ subspaces. Then the average searching size in $\Omega$, is $N_i = \frac{\lceil \lambda N_{\text{seg}} \rceil^k}{N_{\text{sub}}}$. When $N_{\text{seg}} \to \infty$, we have $\frac{\lceil \lambda N_{\text{seg}} \rceil^k}{N_{\text{sub}}} = \lambda$, and $N_i = \lambda^k N_{\text{sub}}$. According to the probability based forwarding strategy in Section 4.3, the average searching size is $N_{\text{prob}} \leq \frac{1}{t} \sum_{i=1}^{t} N_i$. That is, $N_{\text{prob}} \leq \frac{N_{\text{sub}}}{t} \sum_{i=1}^{t} \lambda^k$.

According to the result of Theorem 1, the average searching size $N_{\text{prob}}$ decreases with the reduction of $\lambda$. That is, smaller $\lambda$ brings less matching time in HPartition. Fortunately, the subscriptions distribution in real world applications is often skewed, and most predicates occupy small ranges, which guarantees small average searching size. Besides, note that $\frac{N_{\text{sub}}}{t} \sum_{i=1}^{t} \lambda^k$ reaches its minimal value $N_{\text{sub}} \lambda^k$ if $k_i = k_j$, where $i \in [1, t]$ and $j \in [1, t]$. It indicates that the upper bound of $N_{\text{prob}}$ can further decrease to $N_{\text{sub}} \lambda^k$ if each group has the same number of dimensions.
5 Peer Management

In SREM, there are mainly three roles: clients, brokers, and clusters. Brokers are responsible for managing all of them. Since the joining or leaving of these roles may lead to inefficient and unreliable data dissemination, we will discuss the dynamics maintenance mechanisms used by brokers in this section.

5.1 Subscriber Dynamics

To detect the status of subscribers, each subscriber establishes affinity with a broker (called home broker), and periodically sends its subscription as a heartbeat message to its home broker. The home broker maintains a timer for its every buffered subscription. If the broker has not received a heartbeat message from a subscriber over $T_{out}$ time, the subscriber is supposed to be offline. Next, the home broker removes this subscription from its buffer and notifies the brokers containing the failed subscription to remove it.

5.2 Broker Dynamics

Broker dynamics may lead to new clusters joining or old clusters leaving. In this section, we mainly consider the brokers joining/leaving from existing clusters, rather than the changing of the cluster size.

When a new broker is generated by its datacenter management service, it firstly sends a “Broker Join” message to the leader broker in its top cluster. The leader broker returns back its top cluster identifier, neighbor lists of all levels of SkipCloud, and all subspaces including the corresponding subscriptions. The new broker generates its own identifier by adding a $b$-ary number to its top cluster identifier and takes the received items of each level as its initial neighbors.

There is no particular mechanism to handle broker departure from a cluster. In the top cluster, its leader broker can easily monitor the status of other brokers. For the clusters of the rest levels, the sampling service guarantees that the older items of each neighbor list are prior to be replaced by fresh ones during the view shuffling operation, which makes the failed brokers be removed from the system quickly. From the perspective of event matching, all brokers in the same top cluster have the same subspaces of subscriptions, which indicates that broker failure would not interrupt the event matching operation if there is at least one broker alive in each cluster.

5.3 Cluster Dynamics

Brokers dynamics may lead to new clusters joining or old clusters leaving. Since each subspace is managed by the top cluster whose identifier is closest to that of the subspace, it’s necessary to adaptively migrate a number of old clusters to the new joining clusters. Specifically, the leader broker of the new cluster delivers its top ClusterID carried on a “Cluster Join” message to other clusters. The leader brokers in all other clusters find out the subspaces whose identifiers are closer to the new ClusterID than their own cluster identifiers, and migrate them to the new cluster.

Since each subspace is stored in one cluster, the cluster departure incurs subscription loss. The peer sampling service of SkipCloud can be used to detect failed clusters. To recover lost subscriptions, a simple method is to redirect the lost subscriptions by their owners’ heartbeat messages. Due to unreliable links between subscribers and brokers, this approach may lead to long repair latency. To this end, we store all subscriptions into a number of well-known servers of the datacenters. When these servers obtain the failed clusters, they dispatch the subscriptions in these failed clusters to the corresponding live clusters.

Besides, the number of levels $m$ in SREM is adjusted adaptively with the change of the broker size $N_b$ to ensure $m = \lceil \log_b(N_b) \rceil$, where $b$ is the number of children clusters of each non-top cluster. Each leader broker runs a gossip-based aggregation service \cite{22} at the global cluster to estimate the total broker size. When the estimated broker size $N^{e}_b$ is enough to change $m$, each leader broker notifies other brokers to update their neighbor lists through a gossip-based multicast with a probability of $1/N^{e}_b$. If a number of leader brokers disseminate the “Update Neighbor Lists” messages simultaneously, the earlier messages will be stopped when it collides with later ones, which ensures all leader brokers have the same view of $m$. When $m$ decreases, each broker removes its neighbor list of the top level directly. When $m$ increases, a new top level is initialized by filling appropriate neighbors from the neighbor lists at other levels. Due to the logarithmic relation of $m$ and $N_b$, only significant change of $N_b$ can change $m$. So, the level dynamics brings quite low traffic cost.

6 Experiment

6.1 Implementation

To take advantage of the reliable links and high bandwidth among servers of the cloud computing environment, we choose the CloudStack \cite{23} testbed to design and implement our prototype. To develop the prototype as modular and portable, we use ICE \cite{24} as the fundamental communication platform. ICE provides a communication solution that is easy to program with, and allows the developers to only focus on their application logic. Based on ICE, we add about 11,000 lines of Java code.

To evaluate the performance of SkipCloud, we implement both SkipCloud and Chord to forward subscriptions and messages. To evaluate the performance of HPetration, the prototype supports different space partitioning policies. Moreover, the prototype provides three different message forwarding strategies, i.e., least subscription amount forwarding, random
forwarding, and probability based forwarding mentioned in Section 4.3.

6.2 Parameters and Metrics

We use a group of virtual machines (VMs) in the CloudStack testbed to evaluate the performance of SREM. Each VM is running in an exclusive physical machine, and we use 64 VMs as brokers. For each VM, it is equipped with four processor cores, 8GB memory, and is connected to Gigabit Ethernet switches.

In SkipCloud, the number of brokers in each top cluster \( \delta \) is set to 2. To ensure reliable connectivity of clusters, the average degree of brokers in each cluster \( D_c \) is 6. In HPartition, the entire subscription space consists of 8 dimensions, each of which has a range from 0 to 500. The range of each dimension is cut into 10 segments by default. For each hotspot, the range of its every dimension is cut into 2 segments iteratively.

In most experiments, 40,000 subscriptions are generated and dispatched to their corresponding brokers. The ranges of predicates of subscriptions follow a uniform distribution a-long the entire range of the corresponding dimension. A subspace is labeled as a hotspot if its subscription amount is over 1,000. Besides, the size of basic subscription subset \( N_0 \) in Section 4.4.2 is set to 500, and the threshold value \( \alpha \) in Section 4.4.3 is set to 0.7. The detail of default parameters is shown in Table 3.

Recall that HPartition divides all dimensions into \( t \) individual spaces \( \Omega_i \), each of which contains \( k_i \) dimensions. We set \( k_i \) to be 1, 2, and 4, respectively. These three partitioning policies are called HPartition-1, HPartition-2, and HPartition-4, respectively. Note that HPartition-1 represents the single-dimensional partitioning (mentioned in Section 4.1), which is adopted by BlueDove [14]. The details of implemented methods are shown in Table 4, where SREM-k_i and Chord-k_i represent HPartition-k_i under SkipCloud and Chord, respectively. In the following experiments, we evaluate the performance of SkipCloud through comparing SREM-k_i with Chord-k_i, and evaluate the performance of HPartition through comparing HPartition-4 and HPartition-2 with the single-dimensional partitioning in BlueDove, i.e., HPartition-1. Besides, we do not use APartition (mentioned in Section 4.1) in the experiments, which is mainly because its fine-grained partitioning technique leads to extremely high memory cost.

We evaluate the performance of SREM through a number of metrics.

- **Subscription Searching Size**: The number of subscriptions that need to be searched on each broker for matching a message.
- **Matching rate**: The number of matched events per second. Suppose the first event is matched at the moment of \( T_1 \), and the last one is matched at the moment of \( T_2 \). Thus, the matching rate is \( \frac{N_c}{T_2 - T_1} \).
- **Event loss rate**: The percentage of lost events in a specified time period.

6.3 Space Partitioning Policy

The space partitioning policy determines the number of searching subscriptions for matching a message. It further affects the matching rate and workload allocation among brokers. In this section, we test three different space partitioning policies: HPartition-1, HPartition-2, and HPartition-4.

Figure 6 (a) shows the cumulative distribution function (CDF) of subscription searching sizes on all brokers. Most subscription searching sizes of HPartition-1 are smaller than the threshold value of \( N_{\text{hot}} \). In Figure 6 (a), the percentage of “cold” subspaces in HPartition-4, HPartition-2, and HPartition-1 is 99.7%, 95.1% and 83.9%. As HPartition-4 splits the entire space into more fine-grained subspaces, subscriptions fall into the same subspace if they are interested in the same range of 4 dimensions. In contrast, since HPartition-1 or HPartition-2 splits the entire space into less fine-grained subspaces, subscriptions are dispatched to the same subspace with a higher probability. Besides, the CDF of each approach shows a sharp increase at the subscription searching size of 500, which is caused by SPartition in Section 4.4.2.

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_0 )</td>
<td>64</td>
</tr>
<tr>
<td>( \delta )</td>
<td>2</td>
</tr>
<tr>
<td>( D_c )</td>
<td>6</td>
</tr>
<tr>
<td>( k_i )</td>
<td>{1, 2, 4}</td>
</tr>
<tr>
<td>( N_{\text{seg}} )</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\text{seg}} )</td>
<td>2</td>
</tr>
<tr>
<td>( N_{\text{sub}} )</td>
<td>40K</td>
</tr>
<tr>
<td>( \sigma_{\text{sub}} )</td>
<td>50</td>
</tr>
<tr>
<td>( N_{\text{hot}} )</td>
<td>1,000</td>
</tr>
<tr>
<td>( N_0 )</td>
<td>300</td>
</tr>
<tr>
<td>( \beta )</td>
<td>0.7</td>
</tr>
</tbody>
</table>

<p>| Table 3: Default Parameters in SREM |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Overlay Partitioning Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>SREM-4</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>SREM-2</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>SREM-1</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>Chord-4</td>
<td>Chord</td>
</tr>
<tr>
<td>Chord-2</td>
<td>Chord</td>
</tr>
<tr>
<td>Chord-1</td>
<td>Chord</td>
</tr>
</tbody>
</table>

<p>| Table 4: List of Implemented Methods |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Overlay Partitioning Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>SREM-4</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>SREM-2</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>SREM-1</td>
<td>SkipCloud</td>
</tr>
<tr>
<td>Chord-4</td>
<td>Chord</td>
</tr>
<tr>
<td>Chord-2</td>
<td>Chord</td>
</tr>
<tr>
<td>Chord-1</td>
<td>Chord</td>
</tr>
</tbody>
</table>

---

Fig. 6: Distribution of subscription searching sizes
Figure 6 (b) shows the average subscription searching size of each broker. The maximal average subscription sizes of HPartition-4, HPartition-2, and HPartition-1 are 437, 666 and 980, respectively. Their corresponding normalized standard deviations (standard deviation divided by the average) of the average subscription sizes of each approach are 0.026, 0.057 and 0.093, respectively. It indicates that brokers of HPartition-4 have less subscription searching size and better workload balance, which mainly lies in its more fine-grained space partitioning.

In conclusion, with the increasing of the partitioning granularity, HPartition shows better workload balance among brokers at the cost of the growth of the memory cost.

6.4 Forwarding Policy and Workload Balance

6.4.1 Impact of Message Forwarding Policy

The forwarding policy determines each message to be forwarded to which broker, which significantly affects the workload balance among brokers. Figure 7 shows the matching rates of three policies: probability based forwarding, least subscription amount forwarding and random forwarding.

As shown in Figure 7 the probability based forwarding policy has highest matching rate in various scenarios. This mainly lies in its better trade-off between workload balance and searching latency. For instance, when we use HPartition-4 under SkipCloud, the matching rate of the probability based policy are 1.27× and 1.51× of that of random policy and least subscription amount forwarding policy, respectively. When we use HPartition-4 under Chord, the corresponding gains of the probability based policy become 1.26× and 1.57×, respectively.

6.4.2 Workload Balance of Brokers

We compare the workload balance of brokers under different partitioning strategies and overlays. In the experiment, we use the probability based forwarding policy to dispatch events. One million events are forwarded and matched against 40,000 subscriptions in the experiments. We use an index of \( \beta(N_b) \) [25] to evaluate the load balance among brokers, where

\[
\beta(N_b) = \frac{\sum_{i=1}^{N_b} Load_i}{(\sum_{i=1}^{N_b} \sum_{j=1}^{N_b} Load_i Load_j)},
\]

and Load is the workload of each broker. The value of \( \beta(N_b) \) is between 0 and 1, and the higher value means better workload balance among brokers.

Figure 9 (a) shows the distributions of the number of forwarding events among brokers. The forwarding event size of SREM-k is less than that of Chord-k, which is because SkipCloud spends less routing hops than Chord. The corresponding values of \( \beta(N_b) \) of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 are 0.99, 0.96, 0.98, 0.99, 0.99 and 0.98 respectively. These values indicate quite balanced workloads of forwarding events among brokers. Besides, the number of forwarding events in SREM is at least 60% smaller than in Chord. Figure 9 (b) shows the distributions of matching rates among brokers. Their corresponding values of \( \beta(N_b) \) are 0.98, 0.99, 0.99, 0.99, 0.97 and 0.89, respectively. The balanced matching rates among brokers is mainly caused by the fine-grained partitioning of HPartition and the probability based forwarding policy.

6.5 Scalability

In this section, we evaluate the scalability of all approaches through measuring the changing of matching rate with different values of \( N_{ub} \), \( N_{subs} \), \( k \) and \( N_{seg} \). In each experiment, only one parameter of Table 3 is changed to validate its impact.

We first change the number of brokers \( N_b \). As shown in Figure 10 (a), the matching rate of each approach increases linearly with the growth of \( N_b \). As \( N_b \) increases from 4 to 64, the gain of the matching rate of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 is 12.8×, 11.2×, 9.6×, 10.0×, 8.4× and 5.8×, respectively. Compared with Chord-k, the higher increasing rate of SREM-k is mainly caused by the smaller routing hops of SkipCloud. Besides, SREM-4 presents the highest matching rate with various values of \( N_b \) due to the fine-grained partitioning technique of HPartition and the lower forwarding hops of SkipCloud.
We change the number of subscriptions $N_{\text{sub}}$ in Figure 10 (b). As $N_{\text{sub}}$ increases from 40K to 80K, the matching rate of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 decreases 61.2%, 58.7%, 39.3%, 51.5%, 51.0% and 38.6%, respectively. Compared with SREM-1, each subscription in SREM-4 may fall into more subspaces due to its fine-grained partitioning, which leads to a higher increasing rate of the average subscription searching size in SREM-4. That’s why the decreasing percentages of the matching rates in SREM-$k_i$ and Chord-$k_i$ increase with the growth of $k_i$. In spite of the higher decreasing percentage, the matching rates of SREM-4 and Chord-4 are $27.4 \times 10^6$ and $33.7 \times 10^6$ that of SREM-1 and Chord-1 respectively, when $N_{\text{sub}}$ equals to 80,000.

We increase the number of dimensions $k$ from 8 to 20 in Figure 10 (c). The matching rate of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 decreases 83.4%, 31.3%, 25.7%, 81.2%, 44.0% and 15.0%, respectively. Similar to the phenomenon of Figure 10 (b), the decreasing percentages of the matching rates in SREM-$k_i$ and Chord-$k_i$ increase with the growth of $k_i$. This is because the fine-grained partitioning of HPatient-4 leads to faster growing of the average subscription searching size. When $k$ equals to 20, the matching rates of SREM-4 and Chord-4 are $8.7 \times 10^6$ and $9.4 \times 10^6$ that of SREM-1 and Chord-1, respectively.

We evaluate the impact of different number of segments $N_{\text{seg}}$ in Figure 10 (d). As $N_{\text{seg}}$ increases from 4 to 10, the corresponding matching rates of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 increases 82.7%, 179.1%, 44.7%, 98.3%, 286.1% and 48.4%, respectively. These numbers indicate that the bigger $N_{\text{seg}}$ brings more fine-grained partitioning and smaller average subscription searching size.

In conclusion, the matching rate of SREM shows a linear increasing capacity with the growing $N_{\text{seg}}$, and SREM-4 presents highest matching rate in various scenarios due to its more fine-grained partitioning technique and less routing hops of SkipCloud.

### 6.6 Reliability

In this section, we evaluate the reliability of SREM-4 by testing its ability to recover from server failures. During the experiment, 64 brokers totally generate 120 million messages and dispatch them to their corresponding subspaces. After 10 seconds, a part of brokers are shut down simultaneously.

Figure 11 (a) shows the changing of event loss rates of SREM when 4, 8, 16 and 32 brokers fail. From the moment when brokers fail, the corresponding event loss rates increase to 3%, 8%, 18% and 37% respectively in 10 seconds, and drop back to 0 in 20 seconds. This recovery ability mainly lies in the quick failure detection of the peer sampling service mentioned in Section 3.1 and the quick reassigning lost subscriptions by the well-known servers mentioned in Section 5.3. Note that the maximal event loss rate in each case is less than the percentage of lost brokers. This is because each top cluster of SkipCloud initially has two brokers, and an event will not be dropped if it is dispatched to an alive broker of its top cluster.

Figure 11 (b) shows the changing of the matching rates when a number of brokers fail. When a number of brokers fail at the moment of 10 second, there is an apparent dropping of the matching rate in the following tens of seconds. Note that the dropping interval increases with the number of failed brokers, which is because the failure of more brokers leads to a higher latency of detecting these brokers and recovering the lost subscriptions. After the handling of failed brokers, the matching rate of each situation increases to a higher level. It indicates SREM can function normally even if a large number of broker fail simultaneously.

In conclusion, the peer sampling service of SkipCloud ensures continuous matching service even if a large number of brokers fails simultaneously. Through buffering subscriptions in the well-known servers, failed subscriptions can be dispatched to the corre-
sponding alive brokers in tens of seconds.

6.7 Workload Characteristics

In this section, we evaluate how workload characteristics affect the matching rate of each approach.

First, we evaluate the influence of different subscription distribution through changing the standard deviation $\sigma_{sub}$ from 50 to 200. Figure 8 (a) shows the matching rate of each approach decreases with the increasing of $\sigma_{sub}$. As $\sigma_{sub}$ increases, the predicates of each subscription occupy larger ranges of the corresponding dimensions, which causes that the subscription is dispatched to more subspaces.

We then evaluate the performance of each approach under different event distributions. In this experiment, the standard deviation of the event distribution $\sigma_{e}$ changes from 50 to 200. Note that skewed events lead to two scenarios. One is the events are skewed in the same way as subscriptions. Thus, the hot events coincide with the hot spots, which severely hurts the matching rate. The other is the events are skewed in the opposite way as subscriptions, which benefits the matching rate. Figure 8 (b) shows how the adverse skewed events affect the performance of each approach. As $\sigma_{e}$ reduces from 200 to 50, the matching rate of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 decreases 79.4%, 63.2%, 47.0%, 77.2%, 59.1% and 52.6%, respectively. Although the matching rate of SREM-4 decreases greatly as $\sigma_{e}$ reduces, it is still higher than other approaches under different $\sigma_{e}$.

We evaluate the performance of each approach under different combinations of dimensions (called subscription patterns) in Figure 8 (c). Until now, all above mentioned experiments generate subscriptions by one subscription pattern, where each predicate of the dimension follows the normal distribution. We uniformly select a group of patterns to generate subscriptions. For the predicates that a subscription does not contain, their ranges are the whole ranges of the corresponding dimensions. Figure 8 (c) shows the matching rate decreases with the number of subscription patterns. As the number of subscription patterns grows from 1 to 16, the matching rate of SREM-4, SREM-2, SREM-1, Chord-4, Chord-2 and Chord-1 decreases 86.7%, 63.8%, 2.3%, 90.3%, 65.9% and 10.3%, respectively. The sharp decline of the matching rate of SREM-4 and Chord-4 is mainly caused by the quick increasing of the average searching size. However, the matching rates of SREM-4 and Chord-4 are still much higher than that of the other approaches.

In conclusion, the matching throughput of each approach decreases greatly as the skewness of subscriptions or events increases. Compared with other approaches, the fine-grained partitioning of SREM-4 ensures higher matching throughput under various parameter settings.

6.8 Memory Overhead

Storing subscriptions is main memory overhead of each broker. Since each subscription may fall into the subspaces that are managed by the same broker, each broker only needs to store a real subscription and a group of its identifications to reduce the memory overhead. We use the average number of subscription identifications that each broker stores, noted by $N_{sid}$, as a criterion. Recall that a predicate specifies a continuous range of a dimension. We restrict a continuous range by two numerals, each of which occupies 8 bytes. Suppose that each subscription identification uses 8 bytes. Thus, the maximal average memory overhead of managing subscriptions on each broker is $16kN_{sub} + 8N_{sid}$, where $k$ is the number of dimensions, and $N_{sub}$ is the total number of subscriptions. As mentioned in Section 4.1, HPShard-$k$, brings large memory cost and partitioning latency as $k_i$ increases. When we test the performance of HPShard-8, the Java heap space is out of memory.

![Fig. 12: Memory Overhead](image)

We first evaluate the average memory cost of brokers with the changing of $N_{sub}$ in Figure 12 (a). The value of $N_{sid}$ of each partitioning technique increases linearly with the growth of $N_{sub}$. As $N_{sub}$ increases from 40K to 100K, the value of $N_{sid}$ in HPShard-4, HPShard-2 and HPShard-1 grows 143%, 151% and 150%, respectively. HPShard-4 presents the highest $N_{sid}$ than that of the other approaches, which is because the number of subspaces that a subscription falls into increases with the decreasing room of each subspace. In spite of the high $N_{sid}$ in HPShard-4, its maximal average overhead on each broker only cost 20.2 MB when $N_{sub}$ reaches 100K.

We then evaluate how the memory overhead of brokers changes with different values of $\sigma_{sub}$. As $\sigma_{sub}$ increases, each predicate of subscriptions spans a wider range, which leads to much more memory overhead. As $\sigma_{sub}$ increases from 50 to 200, the value of $N_{sid}$ in HPShard-4, HPShard-2 and HPShard-1 grows 86%, 60% and 27%, respectively. Unsurprisingly, $N_{sid}$ in HPShard-4 still has largest value. When $\sigma_{sub}$ equals to 200, the maximal average memory overhead of HPShard-4 is 6.2 MB, which brings small memory overhead to each broker.

In conclusion, the memory overhead of HPShard-$k$ increases slowly with the growth of $N_{sub}$ and $\sigma_{sub}$.
if \( k_i \) is no more than 4.

7 Related Work

A large body of efforts on broker-based pub/subs have been proposed in recent years. One method is to organize brokers into a tree overlay, such that events can be delivered to all relevant brokers without duplicate transmissions. Besides, data replication schemes [26] are employed to ensure reliable event matching. For instance, Siena [3] advertises subscriptions to the whole network. When receiving an event, each broker determines to forward the event to the corresponding broker according to its routing table. In Atmosphere [4], it dynamically identifies entourages of publishers and subscribers to transmit events with low latency.

It is appropriate to the scenarios with small-scale of subscribers. As the number of subscribers increases, the over-overlays constructed in Atmosphere probably have the similar latency like in Siena. To ensure reliable routing, Kazemzadeh et al. [5] propose a \( \delta \)-fault-tolerance algorithm to handle concurrent crash failure of up to \( \delta \) brokers. Brokers are required to maintain a partial view of this tree that includes all brokers within distance \( \delta + 1 \). Zhao et al. [6] propose a hybrid network architecture, where a tree overlay and a DHT overlay work together to guarantee the high performance of normal operations and high reliability in the presence of failures. The multi-hop routing techniques in these tree-based pub/subs lead to a high routing latency. Besides, skewed subscriptions and events lead to unbalanced workloads among brokers, which may severely reduce the matching throughput.

In contrast, SREM uses SkipCloud to reduce the routing latency and HPartition to balance the workloads of brokers.

Another method is to divide brokers into multiple clusters through unstructured overlays. Brokers in each cluster are connected through reliable topologies. For instance, brokers in Kyra [7] is grouped into cliques based on their network proximity. Each clique divides the whole content space into non-overlapping zones based on the number of its brokers. After that, the brokers in different cliques which are responsible for similar zones are connected by a multicast tree. Thus, events are forwarded through its corresponding multiple tree. Sub-2-Sub [8] implements epidemic-based clustering to partition all subscriptions into disjoint subspaces. The nodes in each subspace are organized into a bidirectional ring. Due to the long delay of routing events in unstructured overlays, most of these approaches are inadequate to achieve scalable event matching. In contrast, SREM uses SkipCloud to organize brokers, which uses the prefix routing technique to achieve low routing latency.

To reduce the routing hops, a number of methods organize brokers through structured overlays which commonly need \( \Theta(\log N) \) hops to locate a broker. Subscriptions and events falling into the same subspace are sent and matched on a rendezvous broker. For instance, PastryString [9] constructs a distributed index tree for each dimension to support both numerical and string dimensions. The resource discovery service proposed by Ranjan et al. [10] maps events and subscriptions into \( d \)-dimensional indexes, and hashes these indexes onto a DHT network. To ensure reliable pub/sub service, each broker in Meghdoost [11] has a back up which is used when the primary broker fails. Compared with these DHT-based approaches, SREM ensures smaller forwarding latency through the prefix routing of SkipCloud, and higher event matching reliability by multiple brokers in each top cluster of SkipCloud and multiple candidate groups of HPartition.

Recently, a number of cloud providers have offered a series of pub/sub services. For instance, Move [13] provides high available key-value storage and matching respectively based on one-hop lookup [16]. BlueDove [14] adopts a single-dimensional partitioning technique to divide the entire space and a performance-aware forwarding scheme to select candidate matcher for each event. Its scalability is limited by the coarse-grained clustering technique. SEMAS [15] proposes a fine-grained partitioning technique to achieve high matching rate. However, this partitioning technique only provides one candidate for each event and may lead to large memory cost as the number of data dimensions increases. In contrast, HPartition makes a better trade-off between the matching throughput and reliability through a flexible manner of constructing logical space.

8 Conclusions and Future Work

This paper introduces SREM, a scalable and reliable event matching service for content-based pub/sub systems in cloud computing environment. SREM connects the brokers through a distributed overlay SkipCloud, which ensures reliable connectivity among brokers through its multi-level clusters and brings a low routing latency through a prefix routing algorithm. Through a hybrid multi-dimensional space partitioning technique, SREM reaches scalable and balanced clustering of high-dimensional skewed subscriptions, and each event is allowed to be matched on any of its candidate servers. Extensive experiments with real deployment based on a CloudStack testbed are conducted, producing results which demonstrate that SREM is effective and practical, and also presents good workload balance, scalability and reliability under various parameter settings.

Although our proposed event matching service can efficiently filter out irrelevant users from big data volume, there are still a number of problems we need to solve. Firstly, we do not provide elastic resource provisioning strategies in this paper to obtain a good
performance price ratio. We plan to design and implement the elastic strategies of adjusting the scale of servers based on the churn workloads. Secondly, it does not guarantee that the brokers disseminate large live content with various data sizes to the corresponding subscribers in a real-time manner. For the dissemination of bulk content, the upload capacity becomes the main bottleneck. Based on our proposed event matching service, we will consider utilizing a cloud-assisted technique to realize a general and scalable data dissemination service over live content with various data sizes.

ACKNOWLEDGMENTS

This work was supported by the National Grand Fundamental Research 973 Program of China (Grant No.2011CB302601), the National Natural Science Foundation of China (Grant No.61379052), the National High Technology Research and Development 863 Program of China (Grant No.2013AA01A213), the Natural Science Foundation for Distinguished Young Scholars of Hunan Province (Grant No.S2010JJ050), Specialized Research Fund for the Doctoral Program of Higher Education (Grant No.20124307110015).

REFERENCES


Xingkong Ma received the B.S. degree in computer science and technology from the School of Computer of Shandong University, China, in 2007, and received the M.S. degree in computer science and technology from National University of Defense Technology, China, in 2009. He is currently a Ph.D. candidate in the School of Computer of National University of Defense Technology. His current research interests lie in the areas of data dissemination, publish/subscribe.

Yijie Wang received the PhD degree in computer science and technology from the National University of Defense Technology, China, in 1998. Now she is a Professor in the Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology. Her research interests include network computing, massive data processing, parallel and distributed processing.

Xiaoqiang Pei received the B.S. degree and M.S. in computer science and technology from National University of Defense Technology, China, in 2009 and 2011, respectively. He is currently a Ph.D. candidate in the School of Computer of National University of Defense Technology. His current research interests lie in the areas of fault-tolerant storage, cloud computing.