2011 IEEE International Parallel & Distributed Processing Symposium

A Scalable and Elastic Publish/Subscribe Service

Ming Li, Fan Ye, Minkyong Kim, Han Chen, Hui Lei
IBM T.J. Watson Research Center,
Hawthorne NY 10532, USA
Email: {liming, fanye, minkyong, chenhan, hlei}@us.ibm.com

***Abstract*—The rapid growth of sense-and-respond applications and the emerging cloud computing model present a new challenge: providing publish/subscribe as a scalable and elastic cloud service. This paper presents the BlueDove attribute-based publish/subscribe service that seeks to address such a challenge. BlueDove uses a *gossip-based one-hop overlay* to organize servers into a scalable cluster. It proactively exploits skewness in data distribution to achieve high performance. By assigning each subscription to multiple servers through a *multi-dimensional subscription space partitioning* technique, it provides multiple candidate servers for each publication message. A message can be matched on any of its candidate servers with one hop forwarding. The *performance-aware forwarding* in BlueDove ensures that the message is sent to the least loaded candidate server for processing, leading to low latency and high throughput. The evaluation shows that BlueDove has a linear capacity increase as the system scales up, adapts to sudden workload changes within tens of seconds, and achieves multi-fold higher throughput than the techniques used in the existing enterprise and peer-to-peer pub/sub systems.**

***Keywords*-Publish/subscribe System; Cloud Computing**

I. Introduction

Publish/Subscribe (pub/sub) is a commonly used asynchronous communication pattern among application components. Senders and receivers of messages are decoupled from each other and interact with an intermediary—a pub/sub system. A receiver(subscriber) registers its interest in the form of a subscription. Messages are published by publishers to the pub/sub system. The system matches publication messages to subscriptions and delivers messages to interested subscribers using a notification mechanism.

There are several ways for subscribers to specify messages of interest. In its simplest form [9], [5], [20], messages are associated with topic strings and subscriptions are defined as the patterns of the topic string. A more expressive form is attribute-based pub/sub [13], [8], [14], where messages are further annotated with various attributes. Subscriptions are expressed as predicates on the message topic and attributes. An even more general form is content-based pub/sub [17], [6], [19], where subscriptions can be arbitrary Boolean functions on the entire content of messages (e.g., XML documents), not limited to attributes[1]. Attribute-based pub/sub strikes a balance between the simplicity and performance of topic-based pub/sub and the expressiveness of content-based pub/sub. Many large-scale and loosely-coupled applications including stock quote distribution, network management, and environmental monitoring can be structured around an attribute-based pub/sub messaging paradigm. Therefore we focus on the attribute-based pub/sub in this paper.

Recent years have witnessed two significant trends that pose new requirements on pub/sub systems. The first trend is the ever increasing number of sense-and-respond applications that adapt their behavior to events in the cyber or real world, based on continuous readings from potentially a prolific number of logical or physical sensors. One example is smart transportation system using a large number of sensors such as smart-phones and road-side cameras [10]. These sensors publish messages with attributes such as time, location and traffic conditions. Individual drivers can subscribe to messages on congestion and accidents specific to their particular routes. The second trend is the emergence of cloud computing, which offers computation infrastructure, platform, and applications as services to one or more tenant organizations. The cloud computing model eliminates upfront capital costs and in-house operational costs for the tenants, allowing the tenants to incur service costs on an as-needed basis. A cloud further minimizes the costs to tenants by elastically scaling its services based on workloads.

These two trends require that pub/sub becomes a scalable and elastic cloud service. However, building pub/sub as a service entails a number of challenges. First, the service must be extremely scalable to support a large number of subscriptions and high message rates. For example, traffic sensors may publish millions of messages per second while tens of thousands of drivers may register millions of subscriptions. A pub/sub service may even have to accommodate many applications of this kind. Such a level of scalability requirement is unprecedented in traditional enterprise pub/sub systems. Second, a modern pub/sub service must be "elastic" to quickly adapt to workload changes that may happen in a short amount of time. For instance, during rush hours, a huge volume of traffic messages and subscriptions are generated; at night, the volume of both reduces substantially. As mentioned earlier, elasticity is also critical to the low-cost benefit of a cloud-hosted service. Third, the service must be resilient to both server and network failures. The new requirements on workload and

[1] Note that some work [14] considers attribute-based pub/sub as one kind of content based pub/sub.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.119

scalability necessitate the provisioning of a pub/sub service from a very large number (e.g., hundreds) of servers. The service must remain available when some of the servers fail or when not all servers can communicate with each other directly. The large number of servers makes server and network failures a common phenomenon that must be handled effectively.

Existing enterprise pub/sub products are not adequate to meet the above requirements. In these products, servers form a cluster and each client establishes *affinity* with one server by connecting to it directly. Subscriptions are often replicated on all servers, such that any server can match messages and forward them to the interested subscribers. Enterprise pub/sub products are not designed with elasticity in mind because the enterprise typically over-provisions the computation resources to meet the needs of peak workload and does not have financial incentives to un-provision the resources during off-peak hours. The full replication and client-server affinity also impose limits on scalability.

There has been a large body of academic research [9], [7], [11], [14] that uses different overlay techniques [21], [15], [16] to provide pub/sub functionality in a peer-to-peer (P2P) manner. In these systems, each peer node is responsible for a small portion of the whole subscription space. Messages and subscriptions are forwarded to corresponding nodes through multi-hop overlay routing. This strategy targets the adverse conditions in wide-area P2P systems: unreliable links and high node churn rates due to frequent node leave/join. It is inappropriate for a pub/sub service provisioned from a data center, where the network connections are much more reliable and the node membership much more stable than the P2P environment. The multi-hop overlay routing incurs unnecessary processing and network delays and does not take advantage of the well engineered data center environment.

This paper presents the BlueDove attribute-based pub/sub service.[2] BlueDove uses one-hop look-up [12] to organize servers into an overlay. Such an overlay is inherently scalable and tolerant to both server failures and network partitions. Adding nodes to increase the capacity can be accomplished by the overlay in a matter of seconds. To ensure scalable pub/sub matching and high throughput, BlueDove employs *mPartition*, a *multi-dimensional subscription space partitioning* scheme that exploits the skewness of subscriptions. It assigns each subscription to multiple servers, such that each message can be matched on any of its corresponding candidate servers. A *performance-aware message forwarding* technique always forwards messages to the least loaded candidate server for matching, and thus achieves low latency and high throughput.

To the best of our knowledge, BlueDove is the first effort that provides attribute-based pub/sub as a cloud service.[3] We make the following contributions in this paper. We identify the differences in the cloud environment compared to traditional enterprise or peer-to-peer pub/sub systems, and point out the implications on the pub/sub architecture. We propose new techniques including mPartition and performance-aware forwarding, that turn data skewness into an asset for scalability and performance, and combine them with existing technique (i.e., one-hop lookup) to build a prototype for scalable, elastic and fault-tolerant pub/sub service. We have carried out thorough and systematic experiments to validate our design and evaluate the performance of our prototype. The results demonstrate a linear capacity increase as the system scales up, agile response to workload changes, and multi-fold improvement in throughput compared to representatives of alternative approaches.

II. BlueDove System Architecture

A. Attribute-based Pub/Sub Model

BlueDove uses a multi-dimensional attribute-based pub/sub model similar to that of [14], [22]. Given k attributes $\{L_1, L_2, \ldots, L_k\}$, let V^i be the (ordered) set of all possible values of attribute L_i, then $\mathbb{V} = V^1 \times V^2 \times \cdots \times V^k$ is the entire attribute space. The attribute space is a k-dimensional space, and from now on we use the terms dimension and attribute interchangeably. A message is defined as a point in the attribute space, $m = (v^1, v^2, \ldots, v^k) \in \mathbb{V}$. For instance, in a traffic monitoring application, four dimensions may be used to describe a message: longitude, latitude, speed, and timestamp. A subscription is modeled as the logical conjunction of k range predicates, each along a different dimension, $(l^1 \leq v^1 < u^1) \wedge \cdots \wedge (l^k \leq v^k < u^k)$. Alternatively, a subscription can be viewed as a k-dimensional hyper-cuboid $\mathbb{S} = S^1 \times \cdots \times S^k \subset \mathbb{V}$, where $S^i = [l^i, u^i)$. By this definition we say a message m matches a subscription $\mathbb{S}$ if and only if $m \in \mathbb{S}$.

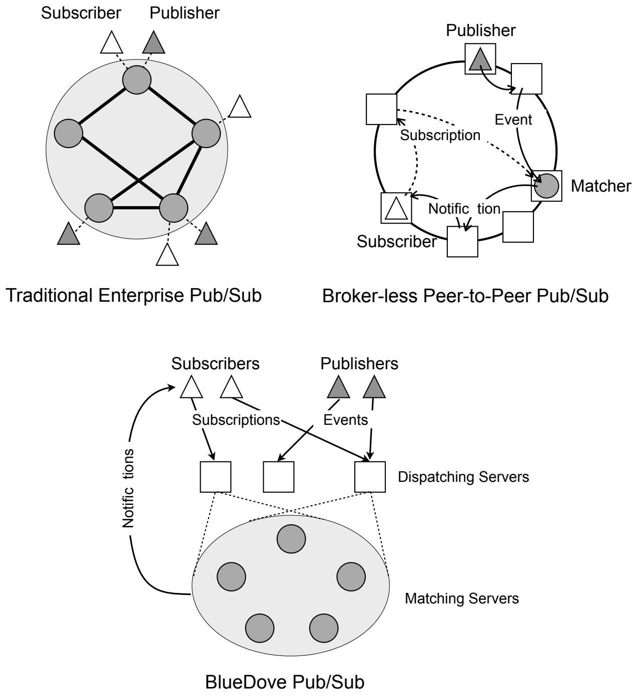
This form of multi-dimensional range query is common in many applications. For example, a driver interested in traffic congestion in a metro area may specify a rectangle covering his proximate area, which can be translated into a subscription, e.g., $[-41 \leq long < -42) \wedge [70 \leq lat < 74) \wedge [0 \leq s < 25)$. The subscription indicates that the driver wants to receive messages where the vehicle speed is in the range of $[0, 25)$ miles per hour, and the vehicle location in a rectangular area with longitude range $[-41°, -42°)$ and latitude range $[70°, 74°)$.

B. System Architecture

Being a cloud-based service, BlueDove operates under different environments than what existing pub/sub systems are designed for. In a traditional enterprise pub/sub system,

[2] BlueDove is actually a cloud-based messaging service that supports both queuing and publish/subscribe. This paper describes only the pub/sub part of BlueDove.

[3] Amazon recently offers a topic-based pub/sub service, but they have not publicized the design or implementation of the system.



Traditional Enterprise Pub/Sub

Broker-less Peer-to-Peer Pub/Sub

BlueDove Pub/Sub

Figure 1: Comparison between three different architectures: enterprise pub/sub, broker-less peer-to-peer pub/sub, and the BlueDove cloud pub/sub.

scalability is achieved by using a small number of brokers, each of which serves a set of locally connected clients (as shown in the top left side of Figure 1). The brokers exchange subscriptions they receive from subscribers, by which they build a content-based routing structure, e.g. in Hermes [14]. The published messages are routed to matching subscribers through the routing structure. Because a client has to receive messages via its locally connected broker (i.e., the so called client-broker *affinity*), the main challenge is how to build states among brokers to route messages towards matching subscribers efficiently. In a cloud-based deployment, the client-broker affinity is no longer required or even desired. This freedom enables more efficient matching and high throughput by grouping "similar" subscriptions to the same server, and limiting routing to one hop.

On the other hand, a broker-less pub/sub system can be constructed using multi-hop overlays for the Internet environment (as shown in the top right side of Figure 1). Subscriptions and messages are routed to certain peers for matching, and messages are delivered to peers with matching subscriptions, all through multihop DHT routing. As discussed in Section I, such a design is motivated by the high node churn and message loss in the P2P environment, which do not apply in a cloud data center. Typical data center servers are connected by high speed local networks. The network delay between servers is small and the packet loss rate is low. Servers stay online for long time unless failures or maintenance happen. Such a well-engineered environment enables BlueDove to adopt the much simpler and faster one-hop lookup [12] for organizing servers.

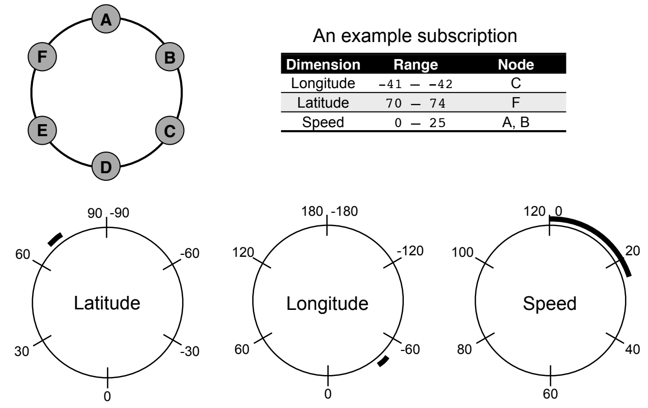
Based on these observations, we propose a two-tier architecture for the BlueDove pub/sub system, as shown in the bottom of Figure 1. A small subset of dispatching servers (called dispatchers) are exposed to the Internet as the "front-end". The IPs of these dispatchers can be publicized through the DNS, and any publisher or subscriber can connect to them directly. They perform light-weight dispatching to send subscriptions and messages to matching servers (called matchers) at the "back-end". For low latency, a message only traverses one hop from a dispatcher to a matcher, where all matching subscriptions are identified. The matcher then delivers messages to subscribers.

The delivery of messages to corresponding subscribers can be direct or indirect. A matcher can send messages directly to matching subscribers if the subscribers can listen and wait for incoming connections or messages. Otherwise, messages can be delivered indirectly: after receiving a subscription from a client, a dispatcher returns a handle to some temporary storage (e.g., a message queue) that the subscriber polls periodically to retrieve matching messages. The matchers only need to deliver messages to the temporary storage. This delivery model is suitable for subscribers such as mobile phones that may not be able to listen on an IP/port waiting for incoming messages.

Such an architecture has a few distinct features that make it suitable for a cloud environment. First, because the elimination of the client-broker affinity, there is no need to build and maintain possibly costly states for content-based routing in a broker network. Instead, a message only goes through very light-weight dispatching and is then matched and delivered to matching subscribers via one matcher. This incurs much less latency and processing compared to multi-hop forwarding in a broker or P2P network. Second, it allows the system to group "similar" subscriptions (e.g., those with same or close predicate ranges) on the same server, so that the local index searching time can be greatly reduced. This is a key factor to the high throughput.

III. BlueDove Component Design

The two-tier architecture addresses how servers should be organized in a cloud pub/sub service. There are still a number of more detailed questions that we need to answer. First, how should we assign subscriptions to matchers such that only one matcher is needed to find all matching subscriptions for any message, while still providing high performance? Obviously, a trivial way is full replication where each subscription is stored on all matchers. But each matcher needs to search through *all* subscriptions for a message, which leads to high latency and low throughput. Second, how should dispatchers decide which matcher to



An example subscription

Dimension	Range	Node
Longitude	-41 – -42	C
Latitude	70 – 74	F
Speed	0 – 25	A, B

Figure 2: An example of mPartition where three dimensions are each split into 6 segments. A matcher is responsible for one segment along each dimension; it stores subscriptions whose predicate ranges overlap with its segments. E.g., the sample subscription's range on latitude $[70 - 74)$ overlaps with segment $[60, 90)$, so it is stored in the corresponding matcher F. Similarly it is stored in C for longitude. In speed dimension its range $[0, 25)$ overlaps with two segments responsible by matchers A and B. So both store a copy of the subscription.

forward a message? Not only do they need to select the *correct* matcher that can find all matching subscriptions, but also the *best* one to achieve high performance. Due to the dynamic changes in both message workload and matcher resource availability, the best matcher for each message differs. In this section, we propose two main components that answer the above questions. They collectively provide high throughput and low latency pub/sub.

- mPartition, a *multi-dimensional subscription space partitioning* component decides how subscriptions are stored on matchers. It assigns each subscription to multiple matchers so that for any message there exist multiple *candidate* matchers, each of which can finish the matching without involving other matchers.
- A *performance-aware message forwarding* component enables dispatchers to always identify the best candidate matchers to achieve high performance despite the dynamic changes in workload and resources.

A. Multi-dimensional Partitioning

In order to take advantage of the many matchers in the system, BlueDove divides the entire subscription space among the matchers so that each matcher only needs to search through a small number of subscriptions. In BlueDove, subscriptions are assigned to matchers using a multi-dimensional subscription space partitioning approach, mPartition. Let N be the total number of matchers. We pick k dimensions in subscriptions as searchable dimensions, where k is an empirical value. In Section IV-F, we evaluate the impact of different values of k. For each searchable dimension L^i, mPartition splits V^i, the set of all possible attribute values on L^i, into N continuous and non-overlapping segments $\{V_j^i, j = 1, \ldots, N\}$. Each matcher M_j is responsible for k such segments V_j^i, one in each dimension. Figure 2 shows a traffic monitoring example, in which each of the three searchable dimensions (longitude, latitude, and speed) are split into 6 segments and assigned to 6 matchers A through F.

Given a subscription $\mathbb{S} = S^1 \times \cdots \times S^k$, a dispatcher assigns $\mathbb{S}$ to matchers k times, each time along a different dimension L^i. It takes the predicate range S^i, then finds which segment(s) $\{V_j^i\}$ overlap with S^i, and forwards a copy of the subscription to corresponding matcher(s). The copy includes not only the predicate S^i on dimension L^i but also all predicates of the subscription. In the example (shown in Figure 2), the subscription's range on latitude dimension $[70 - 74)$ overlaps with segment $[60, 90)$, which is responsible by matcher F. Thus F receives a copy of the subscription. Note that one predicate may overlap multiple segments and more than one matcher may receive the subscription along that dimension (e.g., see dimension *speed* in the example).

Formally, the matcher(s) receiving a subscription along dimension L^i are $\mathbb{M}^i(\mathbb{S}) = \{M_j | V_j^i \cap S^i \neq \varnothing\}$, those whose responsible segments overlap with the predicate range S^i. Since all segments along each dimension cover the whole possible value space, a predicate range has to overlap with at least one segment. Thus the subscription is assigned to at least one matcher in each dimension. In total it is assigned k times to at least k matchers. When a predicate range overlaps with more than one segment in any dimention, the subscription will be assigned more than k times. With this assignment scheme, the entire set of subscriptions $\mathfrak{S} = \{\mathbb{S}\}$ is partitioned among the N matchers k times, each time along a different dimension.

It is possible that the k matchers are not completely different. But its impact on overall performance is limited because the probability of all k copies falling to the same matcher is small. If predicates on different dimensions are distributed uniformly, such probability is $\frac{1}{N^{k-1}}$, with k dimensions and n matchers. In a setup with 4 dimensions and 20 servers, this is about 1 in 8000. In a production cloud data center, N would be much larger and the probability even smaller. If the extreme case happens, BlueDove can explicitly replicate such subscriptions on neighboring matchers, each of which is the clockwise neighbor of the first matcher on a different dimension. With high probability this leads to $(k - 1)$ additional distinct matchers.

Each matcher receives subscriptions along all dimensions. Formally, for each dimension L^i, the subset of subscriptions assigned to matcher M_j is $\mathfrak{S}^i(M_j) = \{\mathbb{S} = S^1 \times \cdots \times S^k | V_j^i \cap S^i \neq \varnothing\}$. A matcher stores subscriptions in each of the k subsets $\mathfrak{S}^i(M_j)$ *separately* and builds a *seperate* index for each subset. This is critical for high performance and will be further explained in Section III-B.

Using a gossip protocol described in Section III-C, the dispatchers maintain a global view of the segment assignment. When a subscription $\mathbb{S}$ enters the system, a dispatcher can find overlapping segments on each dimension and forward $\mathbb{S}$ to the corresponding matchers $\mathbb{M}^i(\mathbb{S})$. Figure 2 shows an example subscription and how it is assigned to 4 matchers along 3 dimensions.

The above mPartition looks similar to the way many DHT overlays partition a ring of ID space into segments and assign them to nodes. However, there are three key differences that confer distinct advantages to BlueDove over enterprise or other DHT-based pub/sub.

1) Multiple Candidate Matchers: Given a message $m = (v^1, v^2, \ldots, v^k)$, mPartition ensures that there exists k matchers. on each of which all matching subscriptions can be found. We call them *candidate matchers* for m. For a dimension L^i, v^i has to fall into one segment V^i_j (i.e., $v^i \in V^i_j$). For any subscription $\mathbb{S}$ that matches m, its predicate S^i must contain v^i (i.e., $v^i \in S^i$). Thus S^i overlaps with V^i_j. By mPartition, $\mathbb{S}$ would have been assigned to the matcher $CM^i(m) = M_j$ responsible for V^i_j. So on M_j all these subscriptions can be found. Note that M_j may also store subscriptions not matching m. They are those whose predicates do not contain m's value on other dimensions.

This enables high performance matching because for each message, any dispatcher can find all its candidate matchers by examining global segment assignment, and forward the message to any of them in just one hop. In contrast, in enterprise or other DHT-based pub/sub [9], [7], [11], [14], messages are forwarded over multiple hops and experience long delays to find all matching subscriptions. Exactly which candidate matcher to choose, is the subject of Section III-B.

2) Exploit data skewness: Real world data distribution is often skewed. The popular "20-80" rule states that 80% of events come from 20% of causes [3]. Some recent study [18] on 7 million users of Twitter (which can be viewed as a topic-based pub/sub system) finds that 80% users (i.e., "topics") have less than 10 followers (i.e., "subscriptions"), while the top 100 popular users each has more than 1.5 million followers.

Although this is not direct evidence for attribute-based pub/sub, we expect similar skewness would appear in the distribution of predicate ranges along some dimensions. Thus some matchers are assigned disproportionately more or less subscriptions along these dimensions than the average (i.e., $|\mathfrak{S}^i(M_j)| \gg$ or $\ll$ average$(|\mathfrak{S}^i(M_p)|)$). Such matchers are on the "hot spots" or "cold spots" of those dimensions. For example (Figure 3), matcher A is at the hot spot of dimension Y and assigned 13 subscriptions on Y, while D is at the cold spot on Y and assigned only 4 subscriptions. During matching they need to search much more or less subscriptions than the average on those dimensions. Note that because a matcher maintains a separate set and index for subscriptions received along each dimension, a matcher may be at the hot spots and cold spots of different dimensions at the same time (e.g., matcher A is on the hot spot of Y but a relatively cold spot on X).

Because k candidate matchers exist for each message along k dimensions, some of them may be on hot spots while others on cold spots. It is desirable to choose those candidates on cold spots, such that much less searching is needed. A message becomes "unlucky" only if all its candidate matchers are on the hot spots of corresponding dimensions, which is unlikely given multiple candidates. We will explain the details in the next subsection.

3) Fault tolerance: Because each subscription has at least k copies stored on different matchers, it provides a natural means of fault-tolerance. If a dispatcher finds that a candidate matcher has failed, or the network connection to that matcher is unavailable, it can forward the message to another candidate. The maximum number of concurrent node failures it can tolerate is simply one less the number of distinct matchers.

B. Performance-aware message forwarding

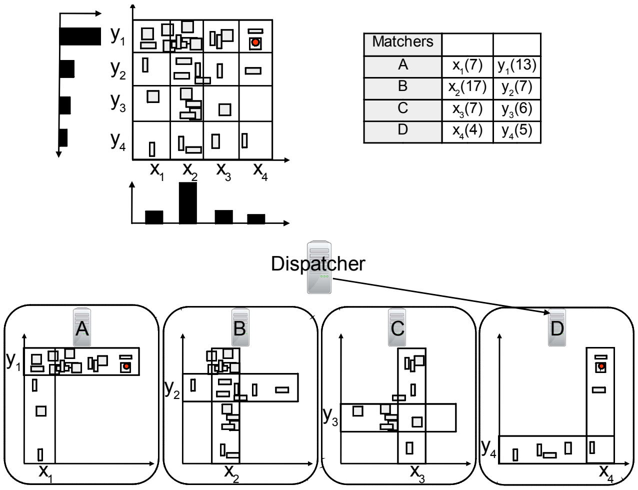
As pointed out previously, given a message m, there are k candidate matchers for it, $CM^i(m)$. Because the amount of subscriptions $|\mathfrak{S}^i(CM^i)|$ and the workload on CM^i may vary greatly depending on the skewness of the subscription distribution, there exists opportunity to choose a cold spot candidate matcher to improve the performance. Here we examine two policies for choosing the best candidate.

1) Subscription amount based policy: The most intuitive way of choosing a candidate is to select the one with the least number of subscriptions on the corresponding dimension. Formally, for a message m, the best candidate matcher is

$$CM(m) = \arg \min_{CM^i(m)} |\mathfrak{S}^i(CM^i(m))| .$$

To this end, each matcher M_j stores subscriptions received along each dimension L^i in a *separate* set $\mathfrak{S}^i(M_j)$. A matcher has k such sets, whose sizes are communicated to all dispatchers using the gossip protocol described in Section III-C. When a dispatcher receives a message m, it first finds the candidate CM^i for each dimension, then it identifies which one has the smallest set of subscriptions on its corresponding dimension. The dispatcher marks that dimension in the message and forwards it to that matcher. The matcher matches the message against the corresponding set of subscriptions only. To avoid interference, a separate queue is used to store incoming messages on each dimension before they are matched.

Figure 3 shows an example where two dimensions are each partitioned into 4 segments and assigned to 4 matchers. The subscriptions (shown as small rectangles) have hot spots



Matchers		
A	x_1(7)	y_1(13)
B	x_2(17)	y_2(7)
C	x_3(7)	y_3(6)
D	x_4(4)	y_4(5)

Figure 3: An example of subscription amount based policy. The table shows the segments and amounts of subscriptions stored by matchers on each dimension. The message (the small dot) falls in the intersection of segments X_4 and Y_1, which are responsible by matcher D and A, respectively. Since D has only 4 subscriptions to search on dimension X, much less than that of A (13 on Y), the message is forwarded to D to match against its subscription set on dimension X.

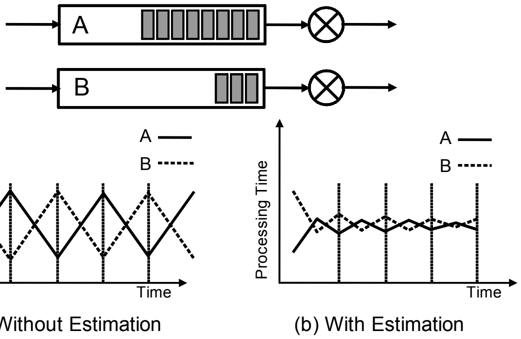
in row Y_1 and column X_2 along the two dimensions. The table shows the segment and the subscription amount for each matcher along the two dimensions. When a message (shown as the dot in the intersection of row Y_1 and column X_4 in the right top corner) arrives, it falls in two segments, row Y_1 and column X_4. The two corresponding matchers D and C are found to have 4 and 13 subscriptions in those segments. D has the smallest amount of subscriptions to search and is chosen to process the message.

The matching throughput is ultimately determined by the total processing time of a message, which includes elements besides matching time. Subscription amount offers only an approximation to the processing time:

- When a matcher is loaded, many messages are waiting in the queue before being matched. The queuing time can be significantly higher than the matching time. This is not reflected by the amount of subscriptions.
- Message matching along different dimensions is another factor. A matcher receives messages, each of which to be matched against one of its k sets of subscriptions. The competition of resources for matching among different sets also affect the matching time of a message, which is not captured in this policy.

Thus this policy does not always achieve optimal performance, which is confirmed by the performance evaluation (see Section IV).

2) Adaptive Policy: To address the deficiencies in the previous policy, we find an adaptive approach that considers the impact of the queue length and competing workloads among different dimensions. Instead of using relatively static subscription amount, it considers the overall *processing time*, which includes both the queuing time and matching time. The message is forwarded to the candidate matcher with the shortest processing time.



(a) Without Estimation

(b) With Estimation

Figure 4: An example of the adaptive policy. Matcher A and B have same matching rate, but different queue length in the beginning. The adaptive policy seeks to keep the total processing time on each matcher the same in order to minimize overall processing time. Plots show the processing time of A and B when dispatcher does and does not estimate queue length of A and B between two updates. Each vertical line represents an update from matchers.

The adaptive policy works as follows. For each dimension L^i, a matcher monitors the message queue length q^i and periodically calculates the average message arrival rate λ^i and matching rate μ^i of the past w seconds. It then sends λ^i, μ^i, and q^i to all dispatchers. The dispatcher estimates the processing time along each dimension for the corresponding candidate matcher, then it forwards the message to the matcher with the shortest estimated processing time.

The estimation is based on linear extrapolation before the next update, assuming that the message arrival and matching rates remain the same between updates. Suppose at last update time t', the queue length on a matcher for dimension L^i was $q^i_{t'}$, the message arrival rate was $\lambda^i_{t'}$, and the message matching rate was $\mu^i_{t'}$. The queue length at time t is calculated as

$$q^i_t = q^i_{t'} + (\lambda^i_{t'} - \mu^i_{t'})(t - t')\,,$$

where $(\lambda^i_{t'} - \mu^i_{t'})(t-t')$ is the number of messages that have arrived but are not processed since t'. Then the time it takes to process the next message is

$$(q^i_t + 1)/\mu^i_{t'}\,,$$

where $q^i_t/\mu^i_{t'}$ is the queuing time and $1/\mu^i_{t'}$ is the matching time.

Figure 4 shows an example of two matchers A and B both processing messages from a dispatcher. [4] Suppose matcher A matches messages much faster than B initially and it has a much smaller total processing time. The dispatcher will forward messages to A and the message queue on A grows quickly. Even before the next update, A's processing time becomes longer than B. Without the extrapolation, the dispatcher will continue to use stale information from the last update, and A's queue will continue to grow while B is idle. Only after receiving the next update can the dispatcher start to redirect messages to B. This causes significant unbalanced load and possibly oscillation among matchers. With the extrapolation, the dispatcher can find out that A's processing time becomes longer than B's before the next update, thus redirecting messages to B promptly. This leads to much more balanced load and avoids oscillation.

In essence, this adaptive policy tries to keep the total processing time the same among all candidate matchers for a given message. This is done by keeping the queue length in proportion to the matching rate of matchers, so that a faster matcher will receive proportionally more messages than a slower one. Another benefit of this policy is that it coordinates multiple dispatchers via the feedback, so that dispatchers do not need to exchange messages explicitly. This makes adding or removing dispatchers much simpler. The leaving or joining dispatcher does not need to calculate or inform other dispatchers how much more or less messages they should forward. They will adjust their forwarding rates automatically via the feedback loop to achieve load balance.

C. *Maintenance of Global State*

In order to perform the multi-dimensional subscription partitioning and message forwarding as described above, BlueDove dispatchers have to maintain a global view of the contact and segmentation information of all matchers. When the number of matchers is large, keeping this global view can be highly expensive. BlueDove employs a gossip-based protocol[12] that exchanges state information among matchers with low overhead. Each matcher maintains a table with the contact information (e.g., IP/port) and segment boundaries (one pair on each dimension) of all matchers, and periodically exchanges the table with $log(N)$ (N is the total number of matchers) randomly selected matchers to keep its table uptodate. Then each dispatcher pulls the table from a randomly chosen matcher once a while to get uptodate view of the global state.

When a new matcher joins the system, it randomly contacts a dispatcher. The dispatcher chooses a heavily loaded matcher, and for each segment on that matcher splits half of the segment to the new matcher. The old matcher then transfers corresponding subscriptions to the new matcher. The segment changes are propagated through the gossip protocol. The leaving of a matcher incurs handing over segments and subscriptions to adjacent matchers, which is simply the reverse of the joining process.

The gossip protocol provides elasticity since adding or removing matchers are handled automatically without manual configuration. It also tolerates node or network failures due to the random selection of nodes to gossip with. Finally, it involves $log(N)$ rounds of gossiping to propagate any state change to the whole network, which is reasonably efficient for large number of matchers. We will present its overhead in Section IV.

IV. Implementation and Evaluation

We have implemented BlueDove based on the code of Cassandra [12], a distributed storage system that handles a large volume of data across many servers. The performance of the prototype is evaluated on a compute cloud platform.

A. *Implementation*

We have added about 20,000 lines of Java code to the Cassandra code base to support new functions such as dispatching and matching for pub/sub. The prototype leverages three existing components in Cassandra: the gossip-based one-hop look-up, the SEDA multi-threading architecture [23], and the event-driven messaging service. The unrelated portion of Cassandra such as the storage service, is disabled so that it does not interfere with BlueDove components. The prototype supports different message forwarding policies, including the subscription amount based policy and the adaptive policy described earlier.

B. *Evaluation Methodology*

We evaluate the performance of BlueDove in a 24-server testbed in the IBM Research Compute Cloud (RC2). Each server is a virtual machine(VM) with four processor cores, 4 GB memory, 60 GB storage, and is connected to Gigabit Ethernet switches. We use 22 VMs in BlueDove: two as dispatchers and the rest as matchers. We run the other two VMs as workload generators; they run simulated applications that generate subscriptions and publish messages. The dispatcher/matcher ratio is 1:10 because we find that event dispatching on dispatchers is almost two orders of magnitude faster than event processing on matchers. Therefore, two dispatchers can handle twenty matchers without becoming the performance bottleneck.

Subscriptions and publications have four attribute dimensions, each of which has a length of 1000. As described in Section II-A, subscriptions are conjunctions of *range* predicates and publications are *points* in the attribute space. In most experiments, $40,000$ subscriptions are generated and they follow a normal distribution with standard deviation of 250. Under such skewness, the density of subscriptions at a hot spot is $2.7\times$ of the average density of subscriptions. The hot spots of dimensions are distributed evenly along the

[4] Only one queue is shown for each matcher for one dimension. In reality, each matcher has multiple queues, one for each dimension.

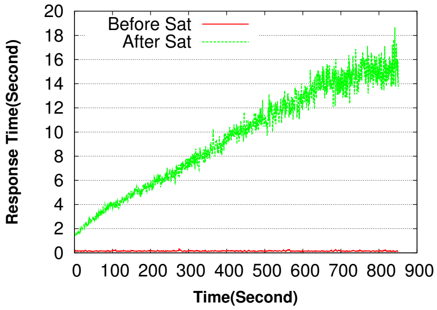


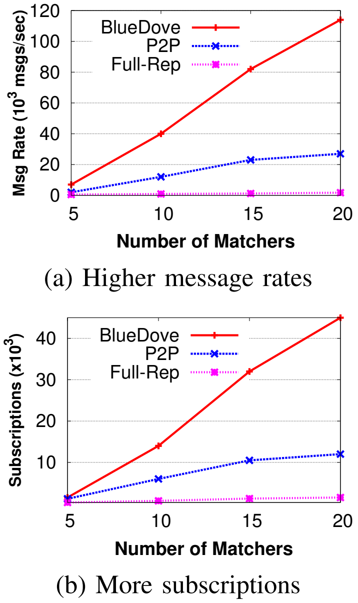
Figure 5: The message response time at the message rate below (100,000 messages/sec) and above (150,000 messages/sec) the saturation rate of 114,000 messages/sec.

full range of 1,000. This is to emulate different skewness between dimensions. The predicate ranges are 250 and the attribute values of messages are distributed randomly in each dimension. In Section IV-F we also evaluate the impact of skewness in subscriptions and publications.

We compare BlueDove against two alternative pub/sub approaches: peer-to-peer (P2P) and full-replication. The P2P pub/sub system builds a peer-to-peer DHT over one dimension of subscriptions and distributes subscriptions to servers through DHT, which is very similar to existing P2P pub/sub systems such as PastryStrings[7] and Sub-to-Sub[22]. In order to ensure fair comparison to BlueDove, our implementation of P2P pub/sub system uses the same gossip-based one-hop overlay as BlueDove. The full-replication approach replicates all subscriptions to all matchers. A message can be forwarded to any matcher to get matched. The difference between P2P and full-replication is how they assign subscriptions and dispatch messages to matchers. In P2P, one dimension is chosen and subscriptions are assigned to matchers based on its predicate on that dimension. There is only one matcher storing each subscription. For each message there is also only one matcher that can match the message. In full replication each matcher stores all subscriptions. Dispatchers simply forward messages to matchers randomly.

We use *response time* and *saturation message rate* as metrics to evaluate pub/sub systems' performance. The response time is the duration from when a message arrives at a dispatcher to when the message is returned to interested subscribers. The saturation message rate is the highest message arrival rate that the pub/sub system can sustain without being saturated. Saturation happens when the message matching speed is lower than the message arrival rate, which results in message queuing and linear growth of response time.

Figure 5 shows the response time as time goes for two message rates, one below and another above saturation. The response time stays constant when the message rate is lower than saturation. The dramatic linear growth when the rate goes above saturation means more and more messages are waiting in the queue. We use this to detect the saturation point by feeding the system with increasing message rates, and checking every five minutes if response time has been growing linearly.



(a) Higher message rates

(b) More subscriptions

Figure 6: Scalability as the number of matchers increases. (a) The saturation message rate with 40,000 subscriptions. (b) The maximum number of subscriptions each system can handle with a message rate of 100,000 messages/second.

C. Scalability

In this section we evaluate BlueDove's scalability of handling high message rates and large number of subscriptions. Specifically we seek to answer two questions: 1) How does the highest sustainable message rate change as the number of matchers increases? 2) How does the highest sustainable number of subscriptions change as the number of matchers increases? We compare BlueDove to the P2P and the full-replication approaches.

We first evaluate how the saturation message rate changes. During each experiment, the workload generators first send 40,000 subscriptions to dispatchers, then they start to publish messages. The message rate is increased every 5 minutes until the system becomes saturated, indicated by monotonically increasing queue lengths and response time. We run the experiment with different numbers of matchers from 5 to 20. Figure 6(a) shows that as the number of matchers increases, the message saturation rate of BlueDove increases much faster than those of the other two approaches. With 5 matchers, the gains of BlueDove over P2P and full-replication are 3.5× and 14×. When the number of matchers increases to 20, these gains became 4.2× and 67×, respectively. In full-replication, adding matchers reduces the number of messages that each matcher needs to handle. But the matching time is not reduced because each matcher needs to search all subscriptions. In both BlueDove and P2P approach, subscriptions are assigned among matchers and thus matching time decreases as more matchers are added. Therefore they handle higher message rates than full-replication. BlueDove also takes advantage of data skewness and chooses the least loaded candidate matcher, which usually needs to search much less subscriptions than average.

We then evaluate how many subscriptions each pub/sub system supports as the system size increases. The workload generators send messages at a constant rate of 100,000 messages/second. Every five minutes, the workload generators send new subscriptions until the system saturates. Figure 6(b) shows that BlueDove handles significantly more subscriptions than the other two approaches. With 20 matchers, the gains of BlueDove are $4\times$ over the P2P approach, and $30\times$ over the full-replication approach.

We also measure the message overhead to maintain the overlay and to update the workload information on dispatchers. The overhead consists of three major components: 1) Using gossip protocol, each matcher exchanges about $2.9K$ bytes of information with a random matcher every second. 2) Each dispatcher pulls the segment table of $60N$ bytes from a random matcher every 10 seconds, where N is the number of matchers. Therefore, the average overhead on each matcher is $60N \times D/10/N = 6D$ byte/second, where D is the number of dispatchers. 3) Each matcher pushes 64 bytes of workload information to each dispatcher every second if the load changes more than 10%. The total overhead on each matcher is about $2.9K + 20D$ byte/second. Even with tens of dispatchers this overhead is only a few thousand bytes per second for each matcher; this is very small for servers connected by Gigabyte switches.

D. Forwarding Policy and Load Balancing

Impact of Message Forwarding Policies: Message forwarding policy determines to which matcher a message is forwarded. It affects the workload allocation among matchers. Figure 7 shows the saturation message rates with 20 matchers for four different policies: the default adaptive policy, the response time based policy (without intrapolation between updates), the subscription amount based policy, and the random policy. The first three policies are performance aware; they select the matcher for each message based on certain performance metrics. The random policy selects a matcher randomly and serves as the baseline for comparison.

The adaptive policy has the highest message saturation rate, which is $1.1\times$ that of the response time based policy, $1.2\times$ of the subscription amount based policy, and $3.5\times$ of the random policy. The adaptive policy estimates the workload on matchers between two consecutive load updates, while the response time based policy does not make such estimates and uses less up-to-date workload information. Thus the adaptive policy outperforms the response time based policy. The subscription amount based policy uses the static workload metric of the number of subscriptions; it does not take into consideration the dynamic changes in queuing and matching times. So it has lower throughput than the previous two policies. Nevertheless, the subscription amount based policy still makes forwarding decision based on some performance metric, thus it achieves $3\times$ the throughput of the random policy.

Load balancing compared to P2P: We compare the load balancing capability of BlueDove with the P2P approach. Because load is uniformly distributed in the full-replication approach, we do not include it in this comparison. We examines the CPU load across different matchers. We use the one-minute average of work load obtained from `/proc/loadavg`[5].

The simulators send 40,000 subscriptions at the beginning of each experiment. Then they send messages slightly below the corresponding saturation message rate to fully load systems without saturating them. Figure 8 shows the CPU load on each matcher. The CPU load on matchers in BlueDove has much less variation than in the P2P approach. Their normalized standard deviations (i.e., standard deviation divided by the average) are 0.14 and 0.82, respectively. This is because the P2P approach forwards messages to matchers without considering load on them, while BlueDove chooses the least loaded matcher.

E. Elasticity and Fault Tolerance

We evaluate BlueDove's elasticity and fault tolerance in this section. Elasticity refers to the system's ability to adapt to sudden changes in workload. It is essential to cloud services since the workload changes continuously. We start the experiments at a small system size of five matchers, an initial message rate of 500 message/second, and 40,000 subscriptions in the system. During the experiments, workload generators increase the message rate by 500 messages/second every five minutes. When a BlueDove dispatcher detects system saturation, it adds a new matcher to distribute the workload. When a new matcher is added, it finds the most loaded matcher in each dimension and takes over about half of its subscriptions.

Figure 9 shows how the response time changes as new servers are added. The three vertical lines show the times when new servers are added. The message response time decreases quickly after new servers are added to the system. The average time from adding a new server to observing a drop in the message response time was 5 seconds. Note that less improvement is observed after adding the third node (the

[5] The load information from `/proc/loadavg` combines CPU load and I/O load. Since matching operations only involve in-memory operations and almost no disk I/O, the load information is a reasonable reflection of CPU load.

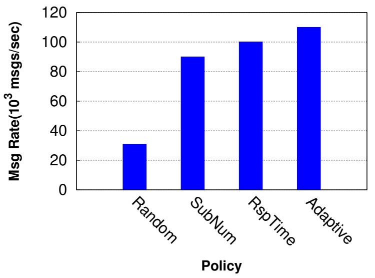


Figure 7: Comparison of different message forwarding policies.

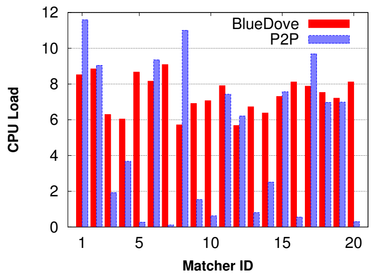


Figure 8: Load balancing. The average CPU load on each matcher.

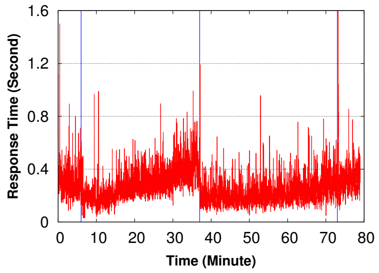


Figure 9: Elasticity. Adaption to increasing message rate.

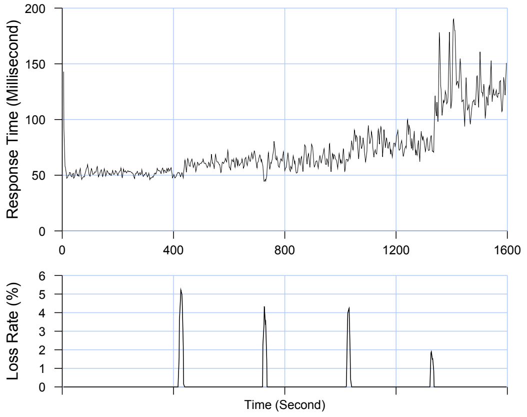


Figure 10: Fault tolerance. The figure on the top shows message response time when servers crash. The vertical lines denote when servers crash. The figure on the bottom shows miss matching rate.

8th node in the system) because it brings relatively smaller increase in the system capacity.

We then evaluate BlueDove's ability to recover from server failures. The experiment starts with 20 matchers. During the experiment, we take one matcher offline every five minutes. Since it takes some time for dispatchers to detect matcher failures, messages sent to failed matchers before detection are lost. The plot on the bottom of Figure 10 shows that the message loss rate increases to around 5% after each matcher failure, but drops back to 0% within 17.5 seconds on average. The plot on the top of Figure 10 shows the message response time over time; the vertical lines indicate matcher failures. The message response time increases slightly after each failure, but the system continues to function normally without saturation.

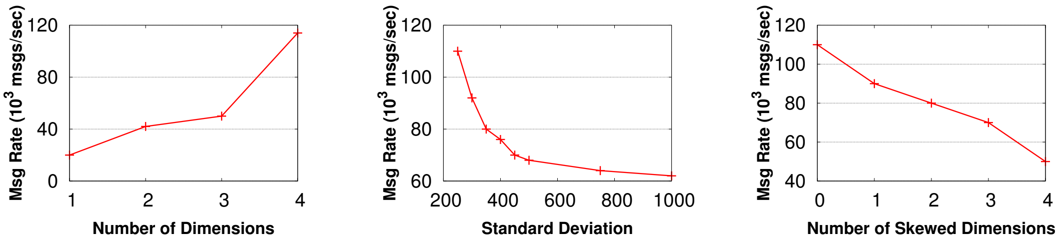
F. Impact of Workload Characteristics

This section evaluates how different workload characteristics may impact the system performance.

Number of Searchable Dimensions: BlueDove relies on the mapping of subscriptions to matchers along multiple dimensions to have the flexibility of multiple candidate matchers to choose from. The flexibility increases with more searchable dimensions. We evaluate the impact of dimensionality on performance. Figure 11(a) shows that the saturation message rate increases as the number of searchable dimensions increases from 1 to 4. The rate with four dimensions is 5.5× as high as that of one dimension. Having more searchable dimensions gives dispatchers more candidate matchers to select from, and thus increasing the probability of avoiding hot spots. Although the maintenance overhead increases with more dimensions, the results show that the gain significantly outweights the overhead.

Skewness of Subscription Distribution: BlueDove takes advantage of the skewness in the distribution of subscriptions to achieve high throughput. What if the subscriptions do not have much skewness? We evaluate how the degree of skewness impacts the performance. We change the standard deviation of the cropped normal distribution of subscriptions to emulate different levels of skewness. The larger the standard deviation, the "flatter" and less skewed the distribution. Figure 11(b) shows the saturation message rate with different standard deviations from 250 to 1000, where the total range of a dimension is 1000. At the standard deviation of 1000, the distribution is quite "flat":the highest number of subscriptions stored by matchers is only 1.17× of the average. Clearly, the message rate keeps decreasing when the standard deviation increases, and drops almost 40% when the standard deviation increased from 250 to 1,000. However, the message rate is still much higher than that of P2P ($\sim$30,000 messages/second, see Figure 6(a)). This shows that as long as some skewness exists, BlueDove can take advantage of it to improve performance.

Skewness of Message Distribution: How does BlueDove perform when messages (publications) are adversely skewed? In all other experiments, messages are distributed evenly in all dimensions. In reality, they can be skewed as well. There are two scenarios: one hurts BlueDove and the other benefits it. First, messages may be skewed in the same way as subscriptions; the hot spots of messages



(a) Searchable dimensions (b) Subscription distribution (c) Message distribution

Figure 11: Workload characteristics. (a) Impact of the number of searchable dimensions. (b) Impact of less skewed subscriptions. (c) Impact of adversely skewed messages.

coincide with the hot spots of subscriptions. These hot spots have many more messages, and they need to be matched against a large number of subscriptions. This decreases the performance. Second, if the hot spots of messages do not overlap with the hot spots of subscriptions, the performance actually improves; compared to uniform distribution, many more messages are matched against matchers with a small number of subscriptions.

We evaluate how skewness in message distribution impacts the performance of BlueDove. Figure 11(c) shows the saturation message rate when the number of dimensions with adverse skewness is increased from 1 to 4. On those dimensions, messages followe the same cropped normal distribution as subscriptions. The rate drops more than 50% when all 4 dimensions are adversely skewed. However, the rate (∼50K) is still much higher than the ∼30K rate of P2P (see Figure 6(a)) with evenly distributed messages. In short, even with adversely skewed message (publication) distribution, BlueDove still outperforms P2P by taking the advantage of the skewed subscription distribution.

V. Related work

There has been a large body of work on pub/sub systems, which can be classified into topic-based, attribute-based, and content-based depending on the matching model. Much early work on pub/sub is topic-based (e.g., Scribe [9], TIBCO [5]). They filter messages based on a single topic string and offer limited expressiveness. Content-based pub/sub systems (e.g., Elvin4 [19], XmlBlaster [17]) are the most expressive ones. They use XPath [17] like subscriptions to match the whole content of messages. Attribute-based pub/sub systems provide more flexibility than topic-based, while requiring less intensive computation than content-based systems. BlueDove falls into the attribute-based category: Each message has values on multiple "dimensions" or "attributes". These values are evaluated against the subscriptions' predicates (e.g., ranges) on these dimensions to identify matching subscriptions. Note that in quite some work [14], [22], attributed-based ones are also called content-based.

In many pub/sub systems, especially those designed for enterprise environment, clients establish *affinity* with brokers, i.e. each client connects to its chosen broker among a connected broker network. A subscriber sends subscriptions and receives matching messages through its chosen broker. Because messages could be published at any broker, brokers need to advertise subscriptions to build routing state in the network. Messages are matched and delivered following the reverse paths to reach matching subscribers. For example, in Siena [8] and Gryphon [13], each subscription is flooded to the whole network. Every broker maintains certain information to determine to which neighboring brokers to forward messages. In Hermes [14], subscriptions and messages of the same topic are sent to a rendezvous server, which uses reverse paths to send back messages to matching subscribers. Compared to them, BlueDove does not have this affinity constraint. We intentionally assign "similar" subscriptions to the same sets of matching nodes (called *matchers*), such that the same matching computation does not need to be repeated on many nodes. Matching a message takes only one matcher, and no reverse path forwarding or computation over multiple intermediate nodes is required.

In other work targeting peer-to-peer environments, publishers and subscribers also serve as matching nodes. There is no dedicated brokers. Such systems usually use structured overlay techniques [21], [15], [16] to deal with dynamics such as node churns, unreliable links and long delay commonly seen in the Internet. Subscriptions and messages are distributed by DHT. For instance, each node in Meghdoot [11] is responsible for a high dimensional cube. Messages and subscriptions are routed through CAN [15] to corresponding nodes, which perform matching. Pastrystrings [7] builds distributed index trees, one in each dimension, over Pastry [16] overlay. Each message has to traverse multiple trees in parallel to find matching subscribers. Yang et al. [24] build multi-dimension search trees(k-d tree) on Chord overlay [21]. In comparison, there is much less dynamics in a cloud data center environment. Thus BlueDove can use much simpler techniques such as one-hop lookup [12] for scalable organization of dedicated servers. Messages and subscriptions are forwarded only one hop before being matched. The new techniques of mPartition and performance-aware forwarding ensure that each message can choose the fastest candidate matcher for best performance.

A number of cloud providers have offered a series of cloud based services such as queuing, storage and database services [2], [4], [12]. The most relevant one to BlueDove is the Amazon Simple Notification Service(SNS [1]) that provides topic based pub/sub service. BlueDove provides a more expressive attribute-based pub/sub, therefore can better support advanced applications. Cassandra[12] is a highly scalable and elastic storage system for cloud applications. BlueDove uses the same one-hop lookup as in Cassandra, but it provides a pub/sub service instead of a storage service.

VI. Conclusions and Future Work

We have presented the BlueDove attribute-based pub/sub service that is intended to support the emerging sense-and-respond applications and the cloud computing model. BlueDove takes advantage of the data center environment to match publications with subscriptions in just one hop. Through multi-dimensional assignment of messages and performance-aware forwarding, it turns the challenge of skewness in data to an asset for high throughput and low latency. Our experiments show that BlueDove can handle workload two orders of magnitude larger than a full-replication approach typically used in existing commercial pub/sub software, and that it can handle workload three times larger than the multi-hop overlay approach commonly used in existing peer-to-peer pub/sub systems. It also adapts to load changes and server failures in a matter of seconds.

We plan to take a number of steps to address the limitations of the current BlueDove and make it more pragmatic and mature. First, due to the failure detection delay, BlueDove may lose a few messages after a server failure. We will add message persistence mechanism to support applications that do not tolerate message loss. Second, when there are large numbers of attributes, using all these dimensions in mPartition can incur significant overhead. Since it is likely that only a small number of attributes are commonly used in subscriptions, we want to study how to identify these attributes and adjust the partitioning accordingly. Third, when certain subscriptions have very wide predicate ranges on some attributes, they will be stored by many matchers along those dimensions. Furthermore, different applications may use different sets of attributes. We will investigate how to alleviate these problems by partitioning the subscription space in a hierarchical manner. One possibility is to divide dispatchers and matchers into different subsets and let them handle different applications. Finally, we also want to compare its performance with a third-party research prototype, to gain better understanding of the trade-offs in load balancing and performance.

References

[1] Amazon simple notification service. http://aws.amazon.com/sns/.

[2] Amazon web services. http://aws.amazon.com/.

[3] Pareto principle. http://en.wikipedia.org/wiki/Pareto_principle.

[4] Windows Azure platform. http://www.microsoft.com/windowsazure/.

[5] Tib/rendezvous. White paper, TIBCO Inc, 1999.

[6] Jboss content based routing. http://docs.jboss.org/jbossesb/docs/4.3.GA/manuals/html/services/ContentBasedRouting.html, 2008.

[7] I. Aekaterinidis and P. Triantafillou. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *ICDCS*, 2006.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3), 2001.

[9] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized publish/subscribe infrastructure. *JSAC, 20(8)*, 2002.

[10] R. Ganti, N. Pham, H. Ahmadi, S. Nangia, and T. Abdelzaher. GreenGPS: A participatory sensing fuel-efficient maps application. In *MobiSys*, 2010.

[11] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Middleware*, 2004.

[12] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *ACM LADIS*, 2009.

[13] J. Nagarajarao, R. E. Strom, , and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, 1999.

[14] P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. In *DEBS*, 2002.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[17] M. Ruff. xmlBlaster: Message oriented middleware (MOM). http://www.xmlblaster.org/xmlBlaster/doc/whitepaper/whitepaper.html, 2000.

[18] E. Schonfeld. On twitter, most people are sheep: 80 percent of accounts have fewer than 10 followers. http://techcrunch.com/2009/06/06/on-twitter-most-people-are-sheep-80-percent-of-accounts-\have-fewer-than-10-follower/.

[19] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin4. In *AUUG*, 2000.

[20] D. Skeen. Vitria's publish-subscribe architecture: Publish-subscribe overview. Technical report, Vitria Technology Inc., 1996.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.

[22] S. Voulgaris, E. Rivi, A. Kermarrec, and M. van Steen. Sub-2-sub: Self organizing content based publish subscribe for dynamic large scale collaborative networks. In *IPTPS*, 2006.

[23] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. In *SOSP*, 2001.

[24] X. Yang and Y. Hu. A DHT-based infrastructure for content-based publish/subscribe services. In *P2P*, 2007.