RBAY: A Scalable and Extensible Information Plane for Federating Distributed Datacenter Resources

Xin Chen*, Liting Hu♭, Douglas M. Blough*, Michael A. Kozuch†, Matthew Wolf‡
*Georgia Institute of Technology, ♭ Florida International University, †Intel Labs, ‡Oak Ridge National Laboratory
xchen384@gatech.edu,lhu@cs.fiu.edu,doug.blough@ece.gatech.edu,michael.a.kozuch@intel.com,wolfmd@ornl.gov

***Abstract*—While many institutions, whether industrial, academic, or governmental, satisfy their computing needs through public cloud providers, many others still manage their own resources, often as geographically distributed datacenters. Spare capacity from these geographically distributed datacenters could be offered to others, provided there were a mechanism to discover, and then request these resources. Unfortunately, single datacenter administrators tend not to cooperate due to issues of scalability, diverse administrative policies, and site-specific monitoring infrastructure.**

This paper describes RBAY, an integrated information plane that enables secure and scalable sharing between geographically distributed datacenters. RBAY's key design features are twofold. First, RBAY employs a decentralized '*hierarchical aggregation tree*' structure to seamlessly aggregate spare resources from geographically distributed datacenters to a global information plane. Second, RBAY attaches to each participating server a '*admin-customized*' handler, which follows site-specific policy to expose, hide, add, remove resources to RBAY, and thus fulfill the task of '*which resource to expose to whom, when, and how*'. An experimental evaluation on eight real-world geo-distributed sites demonstrates RBAY's rapid response to composite queries, as well as its *extensible*, *scalable*, and *lightweight* nature.

I. INTRODUCTION

As computation continues to move into the public cloud, the computing platforms of interest to cloud applications are no longer limited to a single private site, but instead, resemble a 'warehouse' full of nodes assembled from geo-distributed locations. The OpenCirrus [5] initiative, for instance, federated distributed, heterogeneous datacenters from HP, Yahoo!, Intel, CMU, GT, KIT, UIUC, IDA, and others to create a global testbed for participating sites and the broader research community. The GENI [3] project sought to federate fourteen to fifty sites to create a global testbed for network experiments. Microsoft's Pileus system [20] has established ways to guarantee applications' QoS demands, by selectively accessing data from site spanning the world. Much prior work, e.g., Tuba [2], has sought to use such sites to replicate data for increased availability.

However, prior federation efforts did not meet with unqualified success to seamlessly share resources from many small autonomous sites, due in part to the following challenges:

- *Scalability*. When the aggregate number of nodes scales into the thousands, centralized coordinators, such as the master in Ganglia [14], can become system bottlenecks.
- *Diverse Policies*. Autonomous sites have diverse administrative policies for exposing their spare resources.

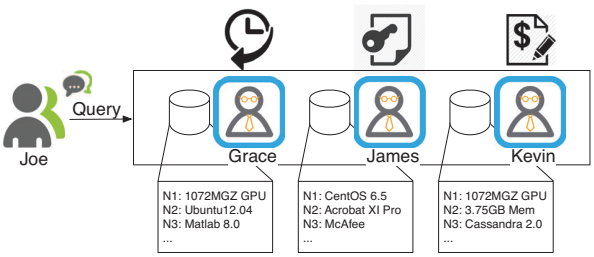


Fig. 1: Motivating usecase scenario.

- *Site-specific Infrastructure*. Individual site uses their own monitoring and management infrastructure (Dell's OpenManage [15], IBM's Tivoli [22], Amazon's CloudWatch [6], etc.), often with internal message transport protocols. Enforcing uniformity across such protocols or infrastructures for seamless operation across global sites will incur considerable cost.

Figure 1 depicts a simplified usecase scenario. Grace, James, and Kevin each administers a collection of nodes with diverse devices they are willing to share, but each sysadmin sets policies for their system independently and may not enjoy any particular admin privileges on the other systems. For example, Grace only wants her resources to be available to others after 10:00 PM; James wants an access control model to expose his resources to users who satisfy certain privacy policies; and Kevin prefers users who have good history logs, e.g., no worrisome behavior.

In this scenario, consider the outside customer Joe is requesting a package of resources for his running. Unfortunately, the inventory information needed to satisfy this request is distributed among the provider Grace, James and Kevin, and managed with different sharing policies — preventing Joe from efficiently locating the resources he needs. Admittedly, authoring all the rights of scheduling resources to a third party may solve the problem, but tracking spare capacity of various resources from many sites and administrating a fair sharing is troublesome and not scalable, especially when the number of nodes or sites increases.

This paper describes RBAY, a public information plane that is (i) ***extensible*** to allow autonomous sites to have diverse administrative policies, (ii) ***scalable*** to the number of resource

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.42

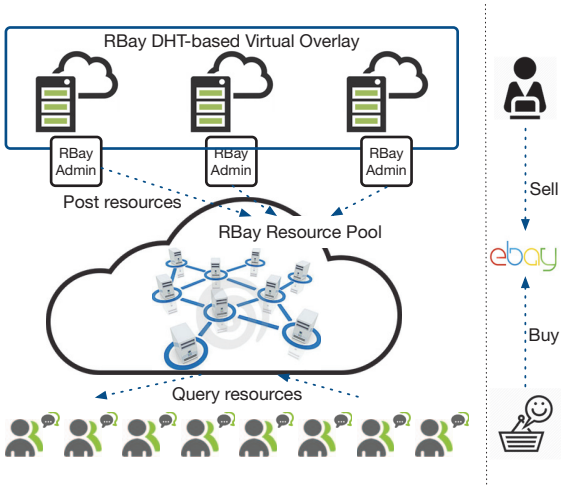


Fig. 2: The RBay software stack akin to eBay.

attributes, nodes and sites, and (iii) ***lightweight*** to each site's existing infrastructure without changing any code.

As shown in Figure 2, RBAY operates in ways akin to 'eBay', where admins '*post*' their resources to the platform, attach certain policy such as '*valid time*', '*password*' and the like. Customers, either from her own site or other sites, set a predicate and query the resources they want. The role of RBAY, therefore, is as an information broker, informing customers about available resources, whereas site administrators retain actual control over resource allocation.

Two key points guide RBAY's design.

Decentralized architecture. RBAY employs a '*hierarchical aggregation tree*' structure based on DHT to seamlessly aggregate spare resources from geographically remote small datacenter site to a global information plane. Then RBAY divides the plane into decentralized subtrees according to different resource attributes, so that each subtree has a partial view of the global information and then the heavy central workload can be balanced to many peers and concurrent queries can be processed by nearby peers in parallel, thus offering attractive scalability property.

Active attribute (AA). Rather than treat a resource attribute as merely a key with a value as prior work did, e.g., ⟨*CPU, Intel 3.40GHz*⟩, RBAY attaches each resource attribute a handler, which is the procedural code written by admins and invoked at runtime. For instance, an admin can write an `if-then` code inside of the `onGet` handler to check whether the password provided by the query equals some predefined password, e.g., ⟨*CPU, Intel 3.40GHz,* `OnGet`⟩. When a customer query performs a `Get` operation on a node, the `onGet` handler is triggered and 'intervenes' to implement password access control for this node's CPU resource.

RBAY's prototype implementation includes a language sandbox for the AA runtime, DHT-based aggregation trees for tracking and storing AAs, and finally, a query interface for searching AAs. Our evaluation with 160 VMs on Amazon's EC2 across eight sites — Virginia, Oregon, California, Ireland, Singapore, Tokyo, Sydney, and Sao Paulo — shows that when processing composite queries in varying numbers of sites, RBAY's response times of around 600 *ms* are comparable to the performance seen for state-of-the-art single site solutions, adding only the delays of cross-site RTTs. Additional simulations suggest that RBAY will continue to perform well, even as datacenter size increases to tens of thousands scale and resource attribute increases to hundreds of thousands.

This paper makes the following technical contributions:

- A decentralized integrated information plane that aggregates cross-site resources (see Sec. II).
- An *active attribute* runtime permitting admins to customize their own management policies (see Sec. III).
- A comprehensive performance evaluation across real-world geo-distributed sites (see Sec. IV).

II. RBAY INFRASTRUCTURE

In this section, we compare the general hierarchical datacenter management infrastructure with RBay's decentralized datacenter management infrastructure and discuss the rationale and benefits behind the design.

A. Previous site management infrastructure

Previous work on datacenter management typically uses hierarchical models, incl. PARMON [4], Supermon [18], CoMon [7], and Ganglia [14]. As illustrated in Figure 3(a), Ganglia, for instance, uses a multicast-based listen/announce protocol for monitoring within clusters and a tree of point-to-point connections to federate cluster-level state. Within a cluster, each node multicasts its local state, so that all cluster nodes have a complete copy of the cluster's state. Multiple clusters' states are aggregated to the tree root, by polling child nodes at periodic intervals. The root is connected to a web front end, which is the major point interacting with admins and serving all posted queries. XML is used for data representation and XDR for data transport.

Although it uses a big hierarchical tree to connect many clusters, the root is still the bottleneck as it maintains the snapshots of all cluster states and becomes the only point to interact with admins and queries. Besides, each site has its own administrative policy to expose resources, giving a heavy burden to the central node to check everyone's policy at each updating period. Therefore, it is not scalable to the number of resource attributes, children nodes and clusters.

In addition, these approaches were based on the premise that all federated clusters should conform to a uniform communication protocol, as done in *Modbus*, *BACnet*, *OPC*, or *SNMP*, to transfer device signals from edge devices to a central device. It is impractical to impose such a solution on many loosely-coupled, heterogeneous, distributed sites.

B. RBay's decentralized Infrastructure

Instead of using centralized infrastructure with one static tree to poll resource updates, RBAY uses a decentralized infrastructure with many dynamic trees, achieving attractive *scalable* and *lightweight* nature, as described next.

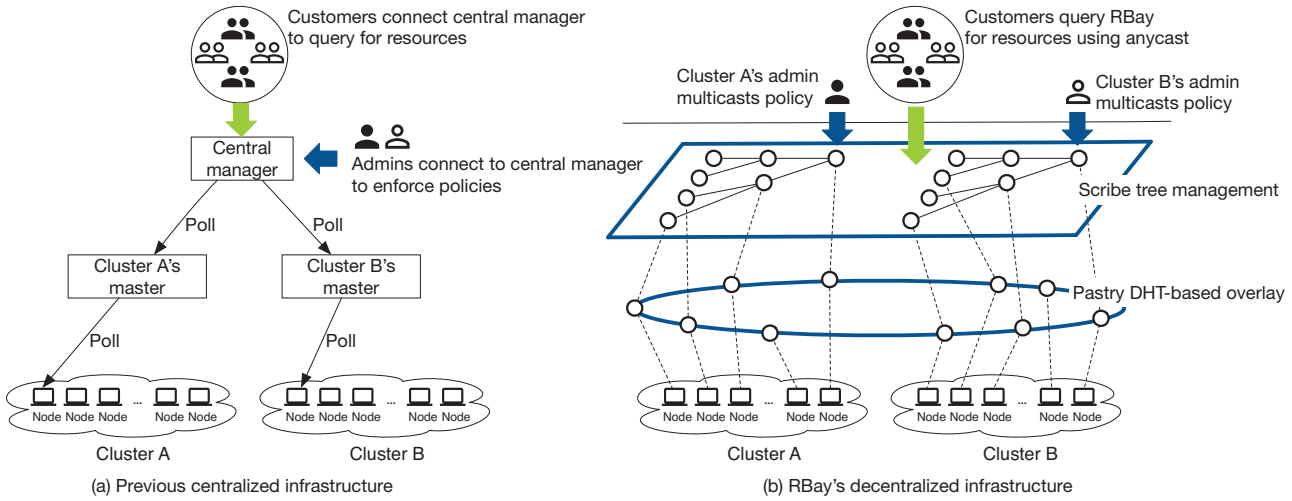


Fig. 3: Comparison of (a) general hierarchical datacenter management infrastructure with (b) our decentralized datacenter management infrastructure.

1) Pastry DHT-based overlay: First, RBAY's participating nodes are self-organized into a Pastry peer-to-peer overlay. In Pastry [17], each node is assigned a unique identifier, 128-bit NodeId that is used to identify nodes and route messages. The set of NodeIds is uniformly distributed; this is achieved by basing the NodeId on a secure **hash** (SHA-1) of the node's IP address.

Given a message and a destination NodeId, Pastry routes the message to the node whose NodeId is numerically closest to the destination within limited $\lceil log_2^b N \rceil$ hops, where b is a base with the typical value 4, and N is the number of nodes.

Routing table. Each Pastry node has two routing structures, a routing table and a leaf set. The routing table consists of node characteristics (IP address, latency information, and Pastry NodeId) organized in rows by the length of common prefix. When routing a message, each node forwards it to the node in the routing table with the longest prefix in common with the destination NodeId.

Leaf set. The leaf set for a node is a fixed number of nodes that have the numerically closest NodeId to that node. This assists nodes in the last step of routing messages and in rebuilding routing tables when nodes fail.

2) Scribe tree management: After nodes are organized into a Pastry DHT-based overlay, RBAY uses an application-level group communication substrate called Scribe [24] to construct many dynamic trees, ensuring nodes having the same attribute can be automatically gathered together to be a tree, e.g., *GPU tree*, *CPU_utilization<10% tree*, *Matlab tree*, and the like.

Scribe builds spanning trees upon Pastry. All nodes join Pastry, and subsequently, nodes may join and leave Scribe trees by their will. Scribe can handle tree sizes varying from one to millions, and it efficiently supports rapid changes in tree membership.

RBAY uses a pseudo-random Pastry Id to name a tree, called TreeId. Usually, the TreeId is the hash of the tree's textual name concatenated with its creator's name. The node whose NodeId is numerically closest to the TreeId automatically becomes the root of the tree.

The tree construction is done as depicted in Figure 3(b). The node satisfying tree's predicate will route a JOIN message towards the TreeId. Since all nodes having the same resource attribute use the same TreeId, their JOIN messages will eventually arrive at a rendezvous node, the root of the tree. The unions of all messages' paths are registered to construct the tree, in which the internal node, as the forwarder, maintains a children table for the tree containing an entry (IP address and NodeId) for each child. Note that the uniformly distributed TreeId ensures the even distribution of trees across all nodes, and thus the load is well-balanced.

3) Multicast, anycast and aggregate: Scribe itself supports two properties: *multicast* and *anycast*. RBAY extends Scribe code to support another important property: *aggregate*.

Multicast. Any node can initiate creating a tree; other nodes can join the tree and then multicast message to all members of the group. Multicast messages are disseminated from the rendezvous point along the multicast tree. RBAY uses multicast to quickly inform members about the admin's policy changes, such as hide or expose available resources, raise or lower rental prices and etc.

Anycast. This is implemented using a distributed depth-first search algorithm of the tree. Any node can anycast to a Scribe tree by routing the message towards the TreeId. Pastry's local route convergence ensures that the message reaches a tree member near the message's sender with high probability. RBAY uses anycast to serve customer query and quickly discover available resources close to the customer.

Aggregate. The states from tree leaves can be periodically 'aggregated' to the tree root within $\lceil log_2^b N \rceil$ hops. All intermediate nodes in the path aggregate the states from their children and progressively roll up the result to the root, in which the

aggregation function can be any composable function, such as `filter`, `sum`, `maximum` or `minimum`, as long as it satisfies the hierarchical computation property [24]. RBAY uses 'aggregate' to calculate a global view of the tree to the root without loading her too much, such as the size of the tree, the average value of all nodes' attributes and etc.

C. Design rationale and benefits

1) Reducing the central computation and I/O bottleneck: As shown from Figure 3(a), in previous management model, all cluster snapshots are flowing to the central master to inform their updates, incurring considerable bandwidth cost. In RBAY, see Figure 3(b), scalable aggregation trees are constructed to take over the central master and each of them only needs to be responsible for one part of cluster snapshot, i.e., one type of resource attribute. Therefore, the central load of tracking availability could be balanced to decentralized peers, reducing the central computation and I/O bottleneck.

2) Decentralized and scalable solution: The TreeId is the hash of the resource attribute's textual name concatenated with its creator's name. The hash is computed using the same collision resistant SHA-1 hash function, ensuring a uniform distribution of TreeIds. Therefore, the tree roots, which are considered the most overloaded nodes, are now uniformed spread over different NodeIds.

Further, because all of these trees share the same set of underlying nodes, each node can be an input leaf, an internal node, the root, or any combination of the above. Besides, the overheads of maintaining a proximity aware overlay network are amortized over all these group spanning trees, resulting in a decentralized solution in which the load is well balanced. Therefore, RBAY is scalable to the number of resource attributes, the number of nodes and sites.

III. RBAY Design and Implementation

RBAY infrastructure offers attractive *scalability* and *lightweight* features. This section describes RBAY's active attribute and prototype implementation which offers attractive *extensible* nature.

A. Software architecture

Figure 4 shows a high-level architecture of a RBAY single node. Each RBAY node consists of three basic components. First is the routing substrate (Figure 4 bottom), which interacts with other RBAY nodes to implement the DHT-based $O(logN)$ mapping.

The second component is the key-value map, which maintains a set of key-value pairs on each node representing resource attributes and their current values. The value can be any type such as *boolean*, *character*, *integer*, *floating-point* and the like, as long as the admin sets and the other site admins approve this setting. We assume that all sites have a uniform way of major resources' key-value pair settings. Examples of these settings are like that $\langle GPU, true\rangle$ indicates that this node has GPU; $\langle CPU, 50\%\rangle$ indicates that this node's average CPU utilization is 50%; $\langle Matlab,$ "9.0"$\rangle$ indicates that this node's Matlab version is 9.0.

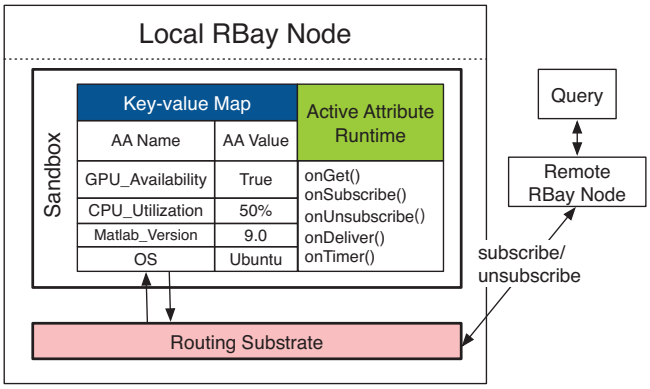


Fig. 4: Local RBay node architecture.

The third architectural component is the active attribute (AA) runtime. The runtime system handles AA invocations and realizes admin's policy on that node. Here is how it works. When a node initially joins RBAY, RBAY assigns it a key-value map (the second RBAY component) which directly reflects resource attribute updates through an underlying monitoring infrastructure (e.g., Libvirt API [12]). Then, a procedure code is associated with this key-value map, which is capable of manipulating the key-value pair's value arbitrarily by admin's will. The code is structured as a set of handlers that specify how the the value changes when certain event occurs, such as the event of expiration of valid sharing time. For example, a node's **`onGet`** handler will be invoked whenever a remote client performs a **`get`** operation to access this node.

The following subsections describe the implementation of the AA runtime (Sec. III-B), the flexible naming scheme (Sec. III-C), the query example (Sec. III-D), also discussed is the additional functionality needed for administrative isolation (see Sec. III-E).

B. Active attribute API and executing environment

Table I shows the handlers run in response to (i) resource lookup events from clients —**`get`**— and (ii) resource management events from admins —**`subscribe`**, **`unsubscribe`**, **`deliver`**.

`OnGet` is the callback function for the **`get`** event. If a query arrives at the node and tries to fetch the NodeId, **`onGet`** handler will be triggered to perform some simple operations on the query, such as password check, history credit check and etc. **`OnSubscribe`**, **`OnUnsubscribe`** and **`onDeliver`** are the callback functions for the **`subscribe`**, **`unsubscribe`** and **`deliver`** events. Periodically (the interval is determined by the **`onTimer`** function), **`onSubscribe`** and **`onUnsubscribe`** are triggered to check if the node belongs to the tree. For example, if it is a *CPU_utilization<10% tree* and the node suddenly becomes overloaded, the node

onGet(callerNode, payload) Invoked when a *get* is performed on the node. Returns a value which will be passed back to the caller. The handler can take an optional *payload* argument of arbitrary type, e.g., user's password, access level.
onSubscribe(callerNode, topic) Invoked upon initial *subscribe* when node joins RBAY and and periodically invoked by RBAY at runtime. Returns the value that determines whether joining the topic tree (e.g., itself or nil). The *topic* argument is usually a string pre-specified by admins.
onUnsubscribe(callerNode, topic) Periodically invoked by RBAY system at runtime. Returns the value that determines whether leaving the topic tree (e.g., itself or nil).
onDeliver(callerNode, payLoad) Invoked upon receiving a control message from administrator. Returns the value that should be updated. It is usually used for administrator to uniformly and interactively manage attribute values. The handler can take an optional *payload* argument of arbitrary type, e.g., new expiration time.
onTimer() Invoked periodically. No return value. It is usually used to perform periodic maintenance such as updating subscription, scheduling to join or leave the system.

TABLE I: Active attribute API.

will unsubscribe the *CPU_utilization<10% tree* at the next interval.

we use Lua [13], a lightweight and easily constrained scripting language, to implement a safe runtime for AAs. Lua technically only has one data structure, a table (an associative array). RBAY represents AAs as Lua table that encapsulates both persistent state and the handlers to be invoked on that state. The name-value mappings in Lua table help us map different events to different handlers. In particular, if the AA table contains an associative array with the names "`onGet`", "`onSubscribe`", "`onUnsubscribe`", "`onDeliver`", or "`onTimer`" and those names are associated with values that are Lua functions, then the runtime invokes those functions when the corresponding events occur. For example, while performing a `get` operation on a certain resource attribute, the runtime system automatically matches it to `onGet` handler. Admins can implement various `onGet` handlers.

For the RBAY runtime, we make several modifications to the standard Lua interpreter. The first is to limit the resource consumption, by strictly limiting the number of bytecode instructions a handler can execute. If a handler exceeds that limit, its execution is immediately terminated. The second is to limit library accesses. The core libraries relating to kernel access, file system access, network access are excluded from the executing environment. As a result, handlers can only do simple math, string, and table manipulation on AA's values.

Figure 5 is a password handler example that restricts the node to be accessible only to certain customers. AA represents the node and it has two properties, *NodeId* and *Password*, saved in AA table. The NodeId or IP address is returned to the customer's query only if the query present some predetermined

```
AA = {NodeId = 27,
      IP = "131.94.130.118",
      Password = "3053482032"}
function onGet(caller, password)
  if (password == AA.Password) then
      return AA.NodeId
  end
  return nil
end
```

Fig. 5: Password handler example.

password, provided as an argument to the `get` operation. Our current implementation simply passes a plaintext password, but can easily be enhanced via encryption primitives involving the AA and public/private key pairs. The node's AA stores the public key, and the query authenticates itself by presenting the corresponding private key.

C. Flexible naming scheme

As we all know, any device includes a batch of properties, such as its type, year of manufacture, version number, manufacturer, model, and the like, e.g., ⟨*Processor Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz 2.39GHz*⟩. If we create an independent aggregation tree for any single property, it indeed has the benefit of simplifying the query model, as no matter what property the customer requires, RBAY can easily forward the query to the specific tree that groups nodes satisfying this property.

However, the disadvantages of this setting are obvious. First, the RBAY platform could be overwhelmed with plenty of unnecessary 'overlapping' trees due to the nest relations among properties. For example, the brands of '*Intel CPU*' and '*AMD CPU*' both belong to '*CPU*', resulting in creating three independent trees in which the last one contains members of the other two. Second, for the purpose of querying resources across sites, federated sites need to maintain a consistent way of resource naming. Maintaining a tree for every new added property means that if a site purchases a cutting-edge device with many new properties, then all sites have to acknowledge the new property names and new trees are created accordingly. Such setting brings about a lot of complexity and inconvenience.

RBAY instead uses a hybrid structure to organize aggregation trees by following the nest relations between properties. For example, the '`model`' trees are subtrees of '`brand`' trees because it is the manufacturer that sets the model and different manufacturers usually have different models. As the '`core_size`' is highly related to the '`model`', so we also make the '`core_size`' trees as the subtrees of '`model`' trees.

To build a hybrid tree structure, RBAY only needs to make a pointer for each subtree root to link to the global root in which each subtree is still a DHT-based flat aggregation tree. By using a hybrid structure, we avoid maintaining unnecessary duplicated aggregation trees.

Furthermore, if a new device with new property is going to be added into the system, the admin only needs to link this new attribute to certain major tree without creating a new aggregation tree. All site admins comply with major trees. Later, when query comes, RBAY query interface parses the query, forwards it to major trees and searches the available nodes recursively. More details will be discussed in subsequent subsection.

D. Query model

RBAY develops a SQL-like query interface based on Zql [25], which takes as input SQL-like queries from nearby clients. Figure 6 shows an example query that finds out k servers from all available sites, satisfying both `CPU_model` equals to "*Intel Core i7*" and `CPU_utilization` is less than "*10%*".

```
SELECT k
FROM *
WHERE CPU_model = "IntelCorei7"
      AND CPU_utlization < 10%
GROUPBY CPU_utlization DESC;
```

Fig. 6: Query example.

RBAY uses the following steps to finish this query (see Figure 7):

- Step 1: The query interface encapsulates two empty messages to send to two different addresses with **NodeIds** equal to **hash("*Intel Core i7*")** and **hash("*10%*")**.
- Step 2: DHT routing guarantees that the root of **tree("*Intel Core i7*")** and the root of **tree("*10%*")** receive the empty messages, fill in the messages with their tree sizes and return the results to the query interface.
- Step 3: After receiving the results, the query interface chooses the tree with smaller size (Let's say **tree("*Intel Core i7*")**) to send another 'anycast' message. This anycast message has a buffer of k empty entries, where k denotes k candidate nodes.
- Step 4: The **tree("*Intel Core i7*")** receives the anycast message and search its members recursively. Each receipt will (i) check if its node has less CPU utilization (<10%); (ii) trigger the AA handler to see if the query has the authorization to obtain the node.
  - If both checks pass, this receipt will reserve the node for the query, fill in one entry of the buffer with its **NodeId** inside the anycast message, and forward the anycast message to the next hop.
  - If not pass, this receipt will do nothing but just forward it to the next hop. The above procedures repeat until k entries have been filled or all tree members have been visited.
- Step 5: After finishing the above steps, the last hop is returning to the query interface, which decapsulates the anycast message and fetches the k **NodeId** out. Then the query interface will commit the nodes for the customer. However, if the customer decides not to take them, the locks on those reserved nodes will be released after a short time window.

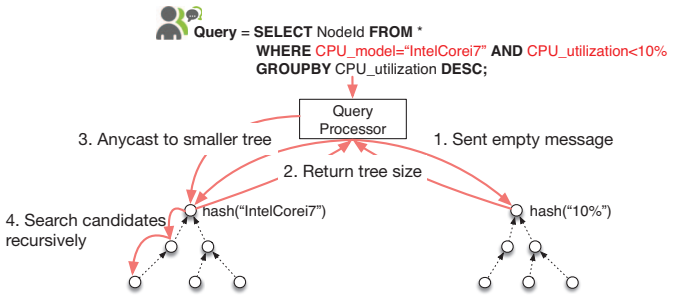


Fig. 7: Steps to handle the sample query.

Each query interface works independently to look up resources for its nearby customers. As a result, if concurrent customers attempt to access the same resource, a conflict occurs when the available resources can only satisfy a portion of customers. The worst case is a 'deadlock' scenario that blocks all customers to access it.

To avoid the 'deadlock' conflict, a truncated exponential backoff time is used for the failed customer to schedule re-query after a failure. The re-query is delayed by an amount of time derived from the slot time and the number of attempts to re-query. After c fails, a random number of slot times between 0 and 2^c-1 is chosen. As the number of failed attempts increases, the number of possibilities for delay increases exponentially. Therefore, if the customer is much more *aggressive* to get more nodes than others, this customer has more possibilities to fail.

E. Administrative isolation

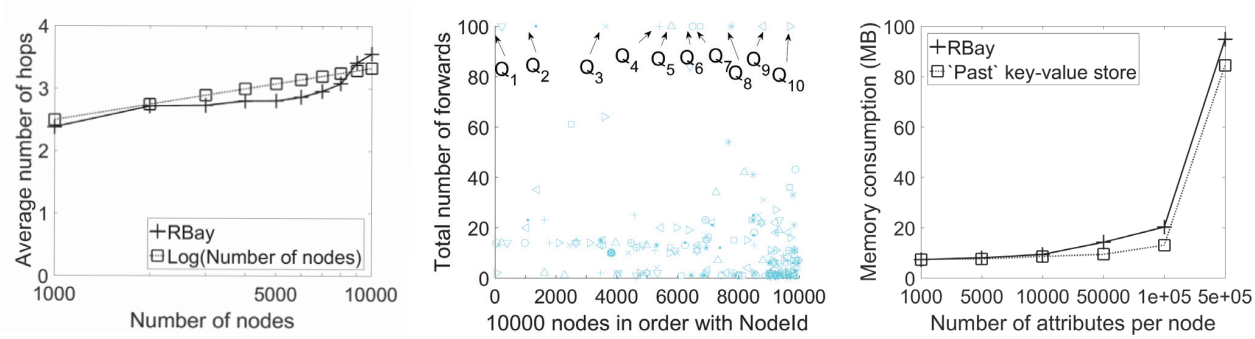
Administrative isolation is important because of (1) security — so that updates and probes flowing in a site are not accessible outside the site, and (2) efficiency — so that site-scoped queries can be locally processed in parallel.

Existing DHTs do not support such site convergence. To route a packet to an arbitrary destination key, the packet can be routed to the destination node in another site as long as it has a longer NodeId prefix matching the key. To ensure site boundaries, like SDIMS [24], we make some simple changes to routing table construction and key-routing protocols. First, each node maintains a separate leaf set, in which each node entry (next hop) is marked with the site to which it belongs. Second, whenever two nodes in a site share the same prefix with respect to a key and no other node in the site has a longer prefix, we introduce a virtual node at the boundary of the site with the prefix plus the next digit of the key, so that all packets towards the key will eventually flow to that virtual node residing on the existing node whose NodeId is numerically closest to the virtual node.

Finally, in order to ensure queries traverse multiple sites to search global resources, we choose certain nodes at the boundaries to act like "*routers*" to route queries across sites.

	Virginia	Oregon	California	Ireland	Singapore	Tokyo	Sydney	Sao Paulo
Virginia	0.559 ms	60.018 ms	83.407 ms	87.407 ms	275.549 ms	191.601 ms	239.897 ms	123.966 ms
Oregon		0.576 ms	20.441 ms	166.223 ms	200.296 ms	133.825 ms	190.985 ms	205.493 ms
California			0.489 ms	163.944 ms	174.701 ms	132.695 ms	186.027 ms	195.109 ms
Ireland				0.513 ms	194.371 ms	274.962 ms	322.284 ms	325.274 ms
Singapore					0.540 ms	92.850 ms	184.894 ms	396.856 ms
Tokyo						0.435 ms	127.156 ms	374.363 ms
Sydney							0.565 ms	323.613 ms
Sao Paulo								0.436 ms

TABLE II: Average round trip latency between Amazon sites.



(a) Scale with #nodes. (b) Scale with #queries. (c) Scale with #resource attributes.

Fig. 8: Scalability evaluation of RBAY with varying number of nodes, concurrent queries, and resource attributes, respectively.

IV. Evaluation

RBAY is evaluated with microbenchmarks and a representative real-world usecase. Experimental evaluations answer the following questions:

- How does RBAY scale with the number of nodes, resource attributes and queries? (Sec. IV-B)
- What are the processing latencies seen for RBAY queries in federated heterogeneous sites? (Sec. IV-C)
- What are the overheads seen for the construction of RBAY aggregation trees in heterogeneous sites? (Sec. IV-D)

A. Testbed and usecase scenario

Experiments are conducted on Amazon EC2 using 160 medium-sized virtual machine instances located at eight sites: Virginia (US East), Oregon (US West), California (US West), Ireland (EU), Singapore (Asia Pacific), Tokyo (Asia Pacific), Sydney (Asia Pacific), and Sao Paulo (South America). Each VM has 2 virtual cores and 4GB of RAM. Each VM runs Ubuntu 14.04 LTS 64 bit, Sun Java SDK 1.6. Unless otherwise noted, queries are issued evenly distributed in all sites. Table II shows the average round trip latencies between pairs of sites.

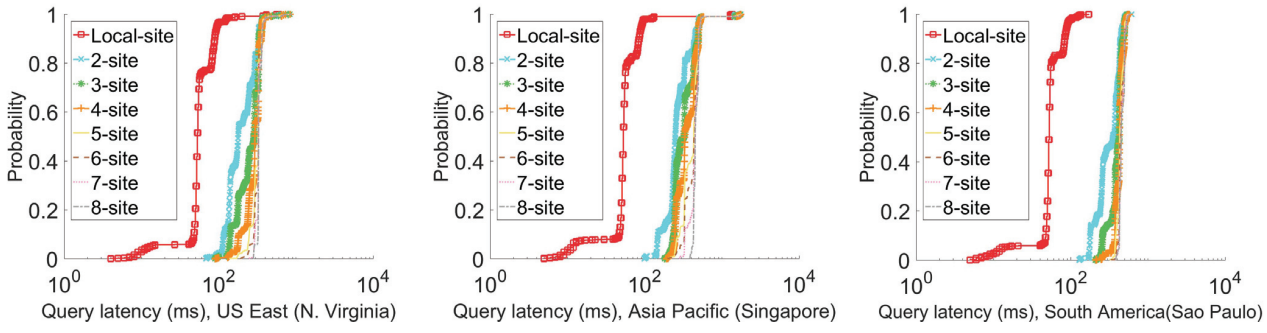
We begin the evaluation with microbenchmark measurements to evaluate RBAY's scalability with the number of simulated agents (JVMs), resource attributes and queries, within each single site. We also evaluate RBAY's memory costs on local node, particularly those pertaining to active attribute (AA) handlers. Microbenchmarks indicate RBAY's *scalable* and *lightweight* nature.

We next evaluate RBAY's *extensible* nature by implementing a password control policy, and evaluate RBAY's tree construction overheads and query processing latencies in heterogeneous sites. We federate Amazon EC2's eight sites into a big resource pool, which consists of 160 VMs, 20 for each site. We launch a total of 16,000 RBAY JVMs to simulate 16,000 RBAY nodes and each node holds 1,000 resource attributes.

Amazon EC2 provides a wide selection of instance types [1] optimized to fit different customer needs. To simulate Amazon EC2's instance family, we create 23 RBAY aggregation trees to represent 23 different instance types in each site. Instance types comprise varying combinations of resource attributes that we mix up randomly. The tree size follows a Gaussian distribution. For example, the center tree of '*c3.8xlarge*' has more members than the edge tree of '*t2.micro*' or '*hs1.8xlarge*'.

In this set of experiments, we sent queries in a speed of 1000 per second to different sites. Each query randomly asks for available nodes holding three random resource attributes and we vary the '*location*' predicate from local single to eight sites. During RBAY runs, the **onGet** handler is invoked for each query to return the NodeId list, only checking if the password matches or not. The **onSubscribe** handler and **onUnsubscribe** handler are invoked to periodically check the current utilization (availability) of each node to determine whether to leave current tree and subscribe to another tree or not.

[1] including *t2.micro*, *t2.small*, *t2.medium*, *m3.medium*, *m3.large*, *m3.xlarge*, *m3.2xlarge*, *c3.large*, *c3.xlarge*, *c3.2xlarge*, *c3.4xlarge*, *c3.8xlarge*, *g2.2xlarge*, *r3.large*, *r3.xlarge*, *r3.2xlarge*, *r3.4xlarge*, *r3.8xlarge*, *i2.xlarge*, *i2.2xlarge*, *i2.4xlarge*, *i2.8xlarge* and *hs1.8xlarge*.



(a) US (N. Virginia) (b) Asia (Singapore) (c) SA (Sao Paulo)

Fig. 9: (a), (b) and (c) show the CDF of latencies for composite queries issued by users in Virginia, Singapore and Sao Paulo respectively, while each query randomly asks for resources from 1-site $\sim$ 8-site (**onGet**).

B. Microbenchmark measurements

We begin the evaluation with microbenchmark measurements to evaluate RBAY's scalability with numbers of nodes, resource attributes and queries within single site. We also evaluate RBAY's memory costs, particularly those pertaining to AAs, and compare to the 'Past' key-value store [16].

*1) **Scale with #nodes**:* We first explore the per query latency by scaling the number of participating nodes. In this experiment, 10,000 RBAY agents are launched to emulate 10,000 heterogeneous nodes, possessing 10 attributes each, while every attribute has a 10% probability to be '**true**' in being exposed to the public for the purpose of sharing. We inject 1,000 atomic queries at a rate of 10 per second, each of which randomly chooses to ask for one unique resource attribute.

Figure 8a shows the average numbers of hops taking by each query to arrive at the destination agent, with varying sizes of datacenter. Result shows that the number of hops increases linearly with an exponential increase in datacenter size. This is because RBAY uses DHT routing protocols for resource discovery, governing hops to be $O(logN)$, where N is the number of nodes in the overlay.

*2) **Scale with #queries**:* Continuing this experiment, to study RBAY's scalability with queries and decentralization in lookup services, we track these 1,000 queries' footprints and present the intermediate nodes' NodeIds that forward these queries. As illustrated in Figure 8b, results illustrate a good load balance among participating agents when performing the routing tasks for the distributed queries. Specifically, since the routing paths of the queries with the same key eventually converge according to DHTs no matter which sources, we can assume the last hop's forwarder to be the node with the highest load. Results show that queries, marked as Q_1, Q_2,...Q_{10}, are evenly distributed across all NodeIds, with an average of 100 forwards. This is because of the independent nature of the resources' keys mapped to different locations in the overlay to divide the central lookup load, resulting in a great potential to host large scale queries.

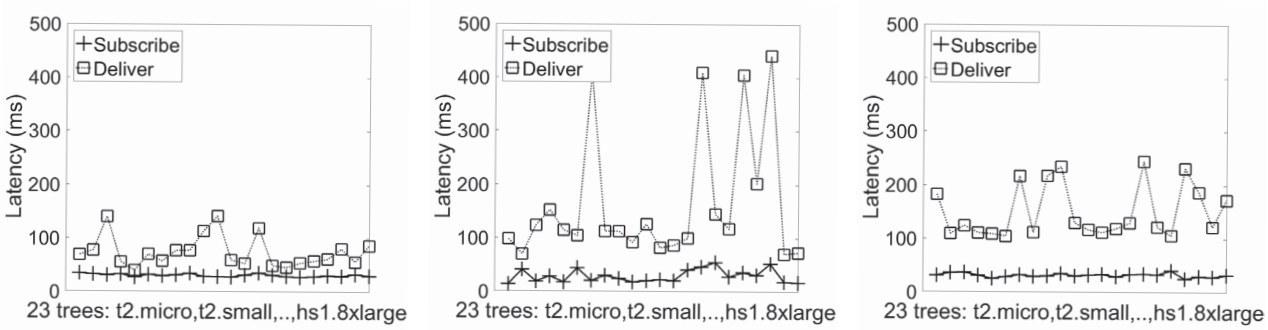
*3) **Scale with #attributes**:* Instead of associating attributes with plain NodeIds, RBAY associates attributes with AAs that consist of NodeIds and admin-specified codes, so it incurs additional memory cost. A natural worry is whether RBAY thereby influences other running applications by occupying excessive memory if the system's attribute space is large. In this experiment, we store an increasing numbers of AAs in the nodes. For RBAY nodes, each attribute is associated with an extra "password" handler besides NodeId, while for Past nodes, only the NodeId is saved, which returns the same list of NodeIds upon a **get** request. Figure 8c shows that when the number of attributes is in the 1000*s*, the difference in memory consumption at this level is negligible (less than 10MB for both RBAY and Past). In the rare case, for 10,000*s* attributes, the overhead relative to RBAY AAs is about 55% to the baseline, but even then the total memory footprint is still reasonable.

C. Query latency

We now shift our attention to the latencies seen by users, by issuing composite queries towards multiple sites, and evaluating how they scale with the number of requesting sites. For these experiments, every site issues 1,000 evenly distributed queries, each of which randomly asks for three attributes focusing on one instance type. We vary the '*location*' predicate from local single to eight sites, asking for return values for the available NodeIds that satisfy the composite query predicate.

Figure 9 shows the CDF for user observed latencies for US's Virginia, Asia's Singapore, and SA's Sao Paulo sites, respectively. For querying a single site, all of the site's users experience comparable latencies. For querying more than one site, the users located in Singapore experience higher latencies, compared to the users located in Virginia and Sao Paulo.

Figure 10 shows overall latency and standard deviation for varying numbers of requesting sites. Results show that as the number of requesting sites increases, average latency increases gradually, and then trends to be stable when for 6 sites, 7 sites, and 8 sites. Generally, it takes less than 200 *ms* for discovering



(a) US (N. Virginia) (b) Asia (Singapore) (c) SA (Sao Paulo)

Fig. 11: Latencies for constructing admin-specified on-demand trees (**onSubscribe**) and latencies for delivering admin-specified commands to tree members (**onDeliver**) in geographically distant sites.

resources in any local site, and it takes around 600 *ms* for searching multiple sites.

The linear increase in latency from 1 site to 5 sites is due to the fact that when a query searches multiple sites, it searches multiple sites in parallel. The user observed latency is mostly limited to the RTT time to the most remote site plus local query time. Whenever the query is issued, if searching 5 sites, it happens to cover all US, EU, Asia, and SA areas over the world, and thus, the latency experiences the worst value that happens to cover the most distant site. That is also the reason why latency tends to stabilize for querying 5∼8 sites, as the max RTT is already included for all.

D. Overhead analysis

Figure 11 shows the comparison of latencies for constructing instance trees and the latencies for delivering admin-specified commands along these trees, for geographically distant sites in the US, EU, Asia, and SA, respectively.

Results show that the latencies of tree construction stabilize around 50 *ms* for all trees and all sites. In contrast, the latencies of command delivery fluctuate; they are 100 *ms* for US and EU sites, but 200∼500 *ms* for the Asia and SA sites. Low latency for **onSubscribe** is due in part to RBAY's fully decentralized overlay in which any RBAY agent only needs to ping its neighboring set to establish a connection for successfully joining the tree, so time cost is not much affected by network conditions. However, **onDeliver** cost is linear with the depth of the tree, which is $O(logN)$.

In this experiment, it may cost 1∼3 hops for the command to reach tree leaves, thus experiencing a higher probability to be interrupted, particularly for sites with unstable networks like the Asia and SA sites.

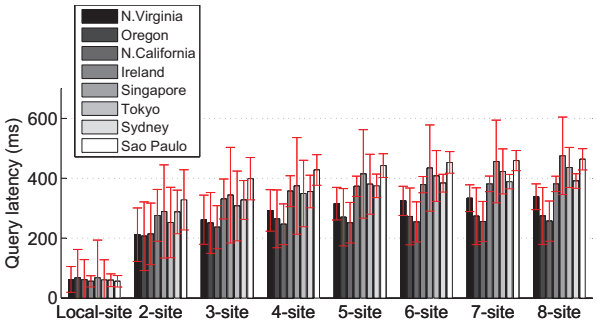


Fig. 10: The average latency and standard deviation for queries issued by users in different locales as the number of involving sites is increased, respectively.

V. Related Work

RBAY includes an extensible local node handler for active attribute and an overlay construction containing dynamic trees for mediating between resource providers and customers. In this section, we compare and contrast RBAY with related work in the literature.

A. Extensible systems

Systems offering extensibility to support application's specific needs have been explored widely. For instance, active networks allow users to inject customized programs into network nodes (e.g., routers) and run the code when nodes are traversed by packets. Database triggers [1] allow applications to define procedural code that is executed in respond to the database operations such as *insert*, *update* and *delete*. Comet [9] is a key-value store that associates each key-value pair with procedural code that is executed in response to storage operations such as *get* and *put*.

B. Overlay construction

Concerning datacenter management tools used for constructing overlay, PlanetLab has management tools like CoMon [7], CoTop [8] and Mon [11]. CoMon and CoTop are web-based general monitors that monitor most PlanetLab nodes. Mon is an on-demand monitoring service that constructs a multicast tree on the fly to serve a one-shot query, but Mon is mainly used for multicasting user commands and has no prior knowledge about the resource attributes. Ganglia [14]

uses a single hierarchical tree to collect all data of federated clusters. GENI [3] uses centralized aggregate manager as a mediator, which aggregates resources from site providers, advertises and allocates resources to experimenters.

The above tools use a relatively centralized infrastructure without in-network aggregation; hence, all individual data are returned to a local machine, even though only their aggregates are of interest. This has limited scalability with the size of the system and the number of attributes.

C. Data aggregation

Astrolabe [23] provides a generic aggregation abstraction and uses a single static tree to aggregate all states. SDIMS [24] uses the same approach but constructs multiple trees for better scalability. Unlike SDIMS, which still assumes a single group for the entire system, Moara [10] maintains many groups for aggregation trees based on different query rates and group churn rates, thus reducing bandwidth consumption. Mariposa [19] is a federated database system which uses an economic paradigm to integrate the data sources into a computational economy to determine the cost, and thus can take into account factors such as resource consumption, staleness of data, supply and demand and etc. In Condor [21], queries and resources advertise themselves to a matchmaker, which is responsible for matching potentially compatible agents and resources.

Different from the above systems, RBAY's novelty lies in '*active*' data aggregation using a fully '*decentralized*' infrastructure with hybrid trees. 'Active' means that admins are granted lots of freedom write their own code inside the callback functions (AA runtime API), as long as it does not violate the executing environment. Admins can also actively interact with RBAY to make changes to their policies by simply reloading the AA table because all callback functions take effect on the next event. 'Decentralized' means that each peer acts like a mediator between queries and resources, which runs in parallel with other peers to make independent decisions.

VI. Conclusion

RBay is a light-weight, non-intrusive and decentralized platform that federates resources and optimizes resource sharing for geographically distributed sites, meanwhile preserving local site autonomy and giving site admins considerable flexibility in specifying their management policies. RBay benefits customers and site admins in several ways. In general, it provides site admins or people who are willing to share their redundant resources a new offering, RBay, in which the resources can be registered easily and admins retain full control over what and how their resources are shared through simple APIs. At the same time, RBay's decentralized architecture also delivers high levels of performance for resource discovery for geographically distributed queries, and low overhead for maintenance. RBay's prototype can be realized in a scalable fashion, without requiring changes to datacenter facilities.

Future work will go beyond additional implementation steps to evaluate RBay's performance under different levels of churn in resources and attribute values, using methods that capture past and predict future churn, based on history, environmental interference, physical location, and other QoS-related or user-relevant factors for AAs. Such factors can also be used to better select appropriate resources in response to user queries, that is, to further optimize the quality of results for queries, including improved consistency, accuracy, and cost-effectiveness.

References

[1] Mysql database triggers. http://dev.mysql.com/doc/refman/5.0/en/triggers.html.

[2] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381. Oct. 2014.

[3] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5–23, Mar. 2014.

[4] R. Buyya. Parmon: A portable and scalable monitoring system for clusters. *Softw. Pract. Exper.*, 30(7):723–739, June 2000.

[5] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. C. Soh. Open cirrustmcloud computing testbed: Federated data centers for open source systems and services research. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09. 2009.

[6] CloudWatch. https://aws.amazon.com/cloudwatch/?nc1=h_ls.

[7] CoMon. http://comon.cs.princeton.edu/.

[8] CoTop. http://codeen.cs.princeton.edu/cotop/.

[9] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13. 2010.

[10] S. Y. Ko, P. Yalagandula, I. Gupta, V. Talwar, D. Milojicic, and S. Iyer. Moara: Flexible and scalable group-based querying system. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 408–428. 2008.

[11] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. Mon: On-demand overlays for distributed system management. In *Proceedings of the 2Nd Conference on Real, Large Distributed Systems - Volume 2*, WORLDS'05, pages 13–18. 2005.

[12] Libvirt. http://libvirt.org/.

[13] Lua. https://www.lua.org/.

[14] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[15] OpenManage. https://www.dell.com/en-us/work/learn/openmanage-essentials.

[16] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 188–201. 2001.

[17] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Confer-*

ence on Distributed Systems Platforms Heidelberg, Middleware '01, pages 329–350. 2001.

[18] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proc. of IEEE Intl. Conference on Cluster Computing*, pages 39–46, 2002.

[19] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1):048–063, Jan. 1996.

[20] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324. 2013.

[21] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[22] Tivoli. https://www.ibm.com/software/tivoli.

[23] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

[24] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 379–390. 2004.

[25] Zql. http://zql.sourceforge.net/.