Hierarchy-Aware Distributed Overlays in Data Centers using DC2

Karthik Nagaraj, Hitesh Khandelwal, Charles Killian, Ramana Rao Kompella
Purdue University
{knagara, hitesh, ckillian, kompella}@cs.purdue.edu

***Abstract*—Popular online services such as social networks, e-commerce and bidding are routinely hosted in large-scale data centers. Group communication systems (*e.g.,* multicast) and distributed key-value stores are among some of the most essential building blocks for these services. Due to their scaling requirements, overlay networks such as distributed hash tables (DHTs) have been traditionally used in such systems. Modern hierarchical datacenter networks and global services running across datacenters pose unique challenges that traditional systems are ill-equipped to handle. For instance, the inherent multi-rooted tree topology design with oversubscription at the core translates into lesser bandwidth at the upper levels of the trees; traditional systems do not take this into consideration leading to a wastage of precious network resources. To solve this problem, we introduce a hierarchy-aware distributed overlay framework called DC2, for large scale and highly dynamic services. We build two applications—DC2-Multicast and DC2-Store—on top of DC2. In our experiments using a real prototype deployed over 700 nodes running over a Modelnet topology with 2 datacenters, we found that DC2-Multicast minimizes message latencies by several orders of magnitude, and reduces node and link stress by a factor of 2 to 3. We also find a reduction in object lookup latency by a factor of 8.**

I. Introduction

In the recent years, we have witnessed an increasing popularity of large-scale online services that run in gigantic datacenters comprising 100s of thousands of machines. Among the many building blocks of these services, two more prominent ones are group communication systems (*e.g.,* multicast trees) and distributed key-value stores. For instance, in social network applications (*e.g.*, Facebook) that comprise millions of users around the world, each user of Facebook can post messages that his group of friends receive in their update feeds, and in turn may receive updates from his friends. Here, group sizes can range from hundreds to a few million members [1] and may change dynamically (as the user makes new friends on a continuous basis). Scalable multicast trees (e.g., Scribe [2]) can be used for fast and efficient distribution of these updates making them a good fit for such applications.

Another common category of applications is electronic commerce and bidding services, in which key-value stores are commonly used to handle dynamic objects in these applications. For example, the current information about a product such as the lowest price and the seller is often stored in a distributed storage system. For large services such as Amazon.com, the total number of products being handled are of the order of millions. There would be a large influx of requests for different products from clients throughout the world, and high throughput and faster response requirements immediately translate into monetary benefits.

Both scalable multicast trees or distributed key-value stores, that form the key building blocks for these datacenter applications, share one basic thing in common: Both rely on a self-organizing overlay network among all the nodes that are part of the distributed system. Ordinarily, for small-scale applications, a centralized approach to constructing the overlay would be sufficient. However, our focus is on large and highly *dynamic* groups with millions of participants, for which a central approach will not scale well. In such cases, Distributed hash tables (DHTs) (*e.g.,* Chord [3], Pastry [4], Bamboo [5], OpenDHT [6]) provide an effective way to constructing these overlay networks with high availability and self-organizing abilities. Indeed, several examples of such systems exist today. For example, Scribe is a multicast system that relies on DHT routing to establish the multicast forwarding trees and can scale to large number of groups and membership dynamics. Similarly, Amazon's key-value store called Dynamo is a distributed storage system built on top of a DHT-like service [7].

Traditional routing in overlay networks focused primarily on optimizing the number of overlay links traversed for each lookup, because they were mainly designed for the wide area with no assumptions about the physical location of the nodes in the network. Thus, they do not pay close attention to several important requirements that are *specific* to datacenter environments. For example, the assumption that each overlay link is pretty much the same is not quite valid in a datacenter context. The physical topology of datacenter networks is often in the form of a multi-rooted tree topology with less overall bandwidth at higher levels of the tree than at the bottom. In datacenter parlance, this is referred to as the over-subscription factor (the ratio of up-links to the down-links), and according to literature, many existing datacenters are heavily oversubscribed with ratios of about 240:1 or 80:1 [8]. This clearly means, the links at the top of the tree are more scarce resources than at the bottom of the tree. However, a straightforward implementation of many existing systems such Pastry [4] or Scribe [2] would pay no heed to this information, and would lead to sub-optimal routes that cross the top of the tree—often many times over.

Moreover, services deployed on a global basis may span many different data centers. In such cases, the links within a

978-1-4673-0298-2/12/$31.00 © 2012 IEEE

datacenter typically cost much less to access than the links that cross datacenters. Even when the service is being hosted by a single entity, it is quite expensive to move data across data centers in different availability zones. Amazon EC2, for instance, charges about 19¢/GB transferred across two datacenters located in different continents, while within US or EU regions, it only charges 1¢/GB [9]. Thus, overlay link costs are not homogeneous and depend a lot on how many datacenters the overlay link spans.

While some overlay systems (*e.g.,* Ammo [10], NICE [11]) did extend beyond the simple hop count as a metric to incorporate latency as one of the metrics to optimize, unfortunately, latency is by itself, not a good determinant in datacenter environments because of close geographical proximity of many of these servers. (Indeed, many datacenter RTTs are in 10s of μseconds and measuring and detecting such low latencies is typically hard.) One could use latency to detect servers across datacenters, but more fine-grained knowledge of the locations of servers is necessary if one wants to take the underlying topology into account. More generally, given the heterogeneity in the link costs and bandwidths, it is important to revisit the design of DHT routing layer to incorporate some notion of hierarchy based on location and/or cost. This hierarchy awareness will help keep communication costs low and minimize the stress on expensive links.

In this paper, we present a framework called DC2 (short for datacenter aware distributed communication) that incorporates hierarchy-awareness in DHT routing. DC2 (discussed in Section II) essentially creates a hierarchy with explicit coordinators for each cluster to coordinate between clusters. This hierarchical overlay helps nodes route group communications locally within a cluster and cross clusters only when necessary. Thus, DC2 helps avoid wasteful extra link traffic within a datacenter, or expensive within-data-center links. To show the generality of DC2, we build two systems on top of DC2—an overlay multicast system called DC2-Multicast and a key-value object store called DC2-Store. We show that these DC2-based systems scale well with large numbers of nodes, and reduce the stress on individual links, particularly the expensive ones, significantly. In our experiments (Section IV) using a real prototype deployed on 700 virtual nodes running over 15 physical machines, we found that DC2-Multicast minimizes message latencies by several orders of magnitude, and reduces node and link stress by a factor of 2 to 3. We also find that the latency of key-value lookups in DC2-Store reduces almost by a factor of 8.

Thus, the main contributions of this paper are as follows:

- We present the design of a first of its kind framework called DC2 for exploiting hierarchical clustering in datacenters to build scalable primitives such as group communications and key-value stores. To the best of our knowledge, datacenter-link awareness has not been considered before in literature for DHT applications.
- We build two applications—DC2-Multicast and DC2-Store—layered over DC2. These applications exploit the DC2 framework for minimizing cost and latency thus increasing the scalability of these applications.
- Using a real prototype, we evaluate these systems for different workloads. We observed that DC2-Multicast builds minimal network-cost trees and DC2-Store provides effective co-located caches.

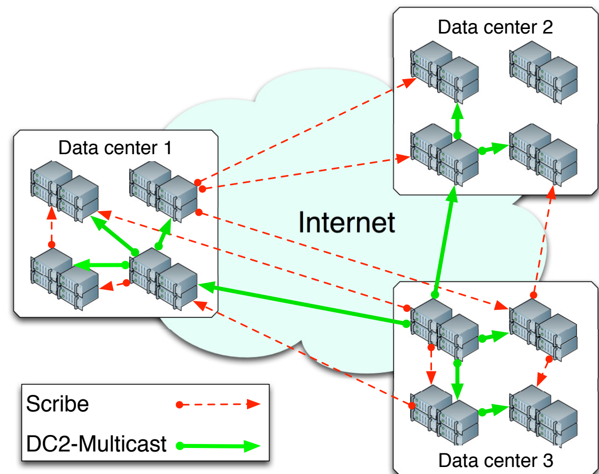

Fig. 1. Comparison of multicast trees constructed across nodes in multiple data centers

II. DC2

In this section, we briefly describe the basic design of the DC2 framework. The DC2 framework builds an overlay that leverages the location of its members, and is scalable to large sizes. It exports a flexible and convenient interface for developing hierarchical location-aware applications, such as DC2-Multicast and DC2-Store illustrated in this paper. DC2 leverages the inherent knowledge of the position of nodes in a datacenter to hierarchically cluster links in order to limit the usage of slower, costly connections to the minimum. Fig. 1 captures the essential gains of DC2-Multicast over another popular location aware tree construction technique called Scribe [2], as observed in several of our experiments that reproduce this scenario. Scribe trees capture the notion of location by attempting to successively reduce the latency of hops in a path from leaf to root, but is not powerful enough to optimize on the long haul links. DC2-Multicast effectively localizes the communication edges within a datacenter before involving the Internet links, reducing the cost of the tree to the lowest possible.

A. Design

The architecture of DC2 involves three major components. First, DC2 relies on the knowledge of the datacenter network topology (or any other hierarchy for that matter) in the form of *location identifiers*. For example, nodes that run DC2 are aware of their location in the datacenter in the form of a *rack id*, *datacenter id*, etc. While such location information is not feasible in a general setting, datacenters are owned and managed by a single authority making it easy to obtain the location information directly. Second, DC2 uses a *scalable overlay routing mechanism* as the key underlying substrate.

Note that DC2 is not tied to any particular overlay routing mechanism—indeed, it can work with any popular routing mechanism. Third, at each level of the hierarchy (*e.g.*, within a rack, within a datacenter), DC2 names one of the nodes in a cluster as a *coordinator*. Nodes route queries towards coordinators at their own level first, followed by routing at the next higher level of the hierarchy and so on, until the appropriate object is located. We describe these in greater detail next.

1) Location identifiers: DC2 constructs clusters out of all end hosts in the network, and hierarchically combines them into a smaller number of clusters at each level. Each end host in DC2 belongs to one of the clusters in the lowest level known as a *leaf-cluster* . Clustering at the next level of the hierarchy groups leaf-clusters into a set of smaller number of *interior-clusters* , where each leaf-cluster belongs to only one interior-cluster. Similarly, clustering continues through many levels of hierarchies (up to h) to terminate in a single interior-cluster. As a real example, in the context of a datacenter network, we can consider a 2-level hierarchy with the lowest cluster formed between servers in the same rack, and the enclosing cluster formed by all racks within the same physical datacenter. The highest level cluster in the 0th hierarchy contains all the datacenter clusters. This clustering is illustrated in Fig. 2, where each node belongs to a single rack cluster, datacenter cluster, and global cluster. The global interior-cluster contains three datacenter interior-clusters, and each datacenter interior-cluster contains two rack leaf-clusters.

The location of an end host is represented as a list of the leaf-cluster and interior-clusters that encompass it. Cluster identifiers are assigned such that no two clusters that share the same parent interior-cluster will share the same identifier. Thus, every leaf-cluster can be uniquely positioned by a list of the identifiers of all its parent interior-clusters. In Fig. 2, the datacenter interior-clusters are numbered 1 through 3 because they share the same global interior-cluster. Within each datacenter cluster, the rack clusters are numbered 1 and 2. Hence, hosts in the left bottom cluster can be identified by the prefix 2*(DC)*:1*(Rack)*:<host id>.

Assigning location identifiers. A key question concerns how to assign location identifiers to each and every host. In a managed datacenters, assigning identifiers is simple, as the placement of all machines and racks is known to the management applications. Therefore finding a mapping from a server machine to a rack, and allocation of unique identifiers to all racks is possible. The identifier for the datacenter can be easily taken from the subnet address (`/16`) block allocated to the datacenter, which will be unique across datacenters anyway. Given the main focus of this paper is largely in the context of enterprises (e.g., Google, Facebook, Yahoo!) that own and manage their datacenters, assigning location identifiers according to the hierarchies is relatively straightforward.

Our hierarchical clustering architecture is not restricted to such managed datacenters alone, however. In multi-tenant cloud environments, where infrastructure providers are often different from the users of the cloud, virtual machines (VMs) hosted on physical machines are leased out to customers. The knowledge of virtual machine to rack mapping is not easily obtainable in such environments. In such environments, we envision that the infrastructure provider, who manages the allocation and maintenance of the VMs, may be able to provide this information to the users as a *service* in future. Note that it is only required for servers to be assigned consistent location identifiers in the form of cluster identifiers, and each server need not know whether any other server is connected to the same rack or not. Further, this location assignment service does not require a high throughput system as the location parameters are only needed during the bootstrap of DC2 (at a server), and can be handled by a single machine.

2) Overlay routing: The next component of DC2 is an overlay routing mechanism. We use Bamboo [5] overlay construction mechanism as the main point of departure. Bamboo is an extension of Pastry [4] with better churn handling and lower initialization costs in our tests. DC2 assigns a unique 160-bit identifier to each node (similar to Bamboo) by constructing a consistent hash of node's IP address and port. A prefix of this identifier is modified as discussed below to infuse the location into the overlay construction. Each node owns a portion of the keyspace between itself and its neighbors on the ring, and every node can route a message to any key in the global key space of this overlay.

Given h-levels in the network structure hierarchy, the location identifiers of a node consist of a list of h cluster identifiers ordered from root to leaf. Each node computes the its modified hash by prefix substituting the list of cluster identifiers in its own hash. For example, let us suppose `[0xe255...]` is the 64-bit hash key based on the IP address and port of the node in a 64-bit hash space. Further, assume that a node that has a 4-bit datacenter id of `0x9` and 4-bit rack id of `0x8`. In this case, the node identifier will be re-written with the first 8 bits `0xe2` replaced by `0x98` corresponding to the concatenation of the datacenter id and the rack id, leading to the hash key `[0x`**98**`55...]` for the node.

The distance between two hash keys in the DHT is defined as the absolute minimum difference of the keys. Nodes in the same cluster will therefore form a contiguous sequence in the logical ring, because they share the same prefix bits. There would also be a longer contiguous sequence of an upper level cluster. This arrangement is shown in Fig. 2 where nodes in the same rack are clustered together in the logical key space, and nodes from the same datacenter form a bigger cluster.

3) Cluster coordinator: DC2 exposes each end host to a cluster coordinator for each key, at each level of the hierarchy, such that all hosts within the same cluster forward their queries directly toward the cluster coordinator in a consistent manner. These coordinators are meant to be rendezvous points at various levels for nodes to congregate, and combine group traffic. The cluster coordinators are identified based on the group key, and hence randomly (yet consistently) load balance the responsibility among nodes on the overlay ring. This is achieved through limited prefix substitution of the group key using the location identifiers. For a hierarchy h', the cluster

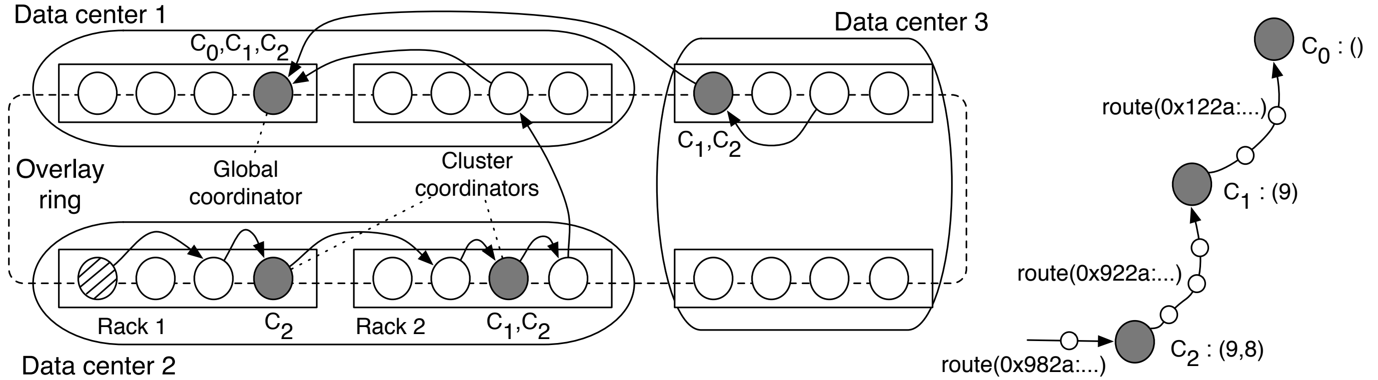

Fig. 2. DC2 overlay ring showing hierarchical clustering of end hosts, with highlighted cluster coordinators and overlay routes showing cluster-routed communication (for a single group).

coordinator is the node who owns the key derived by prefix substituting the first h' identifiers of the identifying node. Since only the nodes at the hierarchy h' of the overlay share this prefix, the cluster coordinator will be a member of this set. A route is established to the coordinator by simply routing on the overlay.

For example, for a topic "Alice's friends" with hash key `[0x122a...]`, if the query originates from a node within datacenter `0x9` and rack `0x8`, we prefix-substitute the object key with `0x98` to obtain the key `[0x`**98**`2a...]`. The node that owns this key lies within the cluster where the actual query originated, and is the cluster coordinator at the rack-level (C_2) for this key. The process of routing to C_2 using the modified key is shown in the right side of Fig. 2, and the individual hops within the rack to reach C_2 are shown in the left. For the cluster coordinator at the datacenter level (C_1), only the datacenter prefix is substituted to obtain `[0x`**9**`22a...]`, and route the query to the node who owns this key. Finally, routing to the original group key terminates at the global coordinator (C_0). Since objects are located consistently using the same hash key, the set of nodes that have the same datacenter id and rack id, will route it to the same cluster coordinator at the rack level, and those with the same datacenter id will route it to the same cluster coordinator at the datacenter level. Fig. 2 exemplifies this where another node in datacenter 3 will route to the same global cluster coordinator for this group. An invariant of such a mechanism to identify coordinators is that a cluster coordinator for hierarchy h' is also a coordinator for all hierarchies $k > h'$ (*e.g.* datacenter coordinators are marked C_1, C_2).

The three basic components embodied in the DC2 architecture provide a powerful framework for location-aware group communication primitives. We now use these primitives to describe two applications – a multicast and a key-value store application, that minimize the number of cross-hierarchy links.

B. Application 1: DC2-Multicast

We derive some underlying principles in constructing multicast trees from Scribe [2], to build efficient trees using overlay routing provided by DC2. Scribe constructs trees using the nodes on the overlay path from a subscriber to the group key owner. In our experiments the locality properties of Scribe were not sufficient in the data center setting, even though Scribe has the potential to scale well. Another important disadvantage of Scribe is the involvement of *forwarder* nodes in each tree to simply forward traffic. This increases the bandwidth footprint of the trees on the whole network, and increases the node stress on each node.

To avoid the pitfalls of Scribe, we also consider another popular tree construction mechanism called *RandTree* that forms the control tree for a number of large-scale distributed systems such as Bullet [12] and RanSub [13]. RandTree constructs a degree-constrained random overlay tree that is typically resilient to node failures and network partitions. A node requests itself to be joined to the tree by starting with one of a few designated nodes. The node that receives a *Join* adopts the new node as a child, provided that it has not run out of children capacity. Supposing that it is already full, the node picks one of its random children to forward the *Join*, and successively this procedure repeats till an empty position is found. This process is guaranteed to terminate if every node can parent at least one child (for each group).

We design DC2-Multicast that adapts the RandTree tree construction protocol within each hierarchical cluster to form smaller sub-trees within a hierarchy. Each multicast group hashes the topic name to obtain a unique hash that will be used for DC2 routing. In DC2-Multicast, each cluster coordinator maintains the knowledge of the root node corresponding to its groups at different levels. A new node that intends to join a given group, would be able to learn the identity of the cluster root node for a group from its cluster coordinator using DC2. Following this, the node can directly contact the root to join the tree local to the cluster, following which the RandTree tree construction protocol described before is executed.

By using hierarchical clustering, we restrict the number of nodes to a small and manageable number ensuring that the Join and Leave operations in the RandTree protocol are low

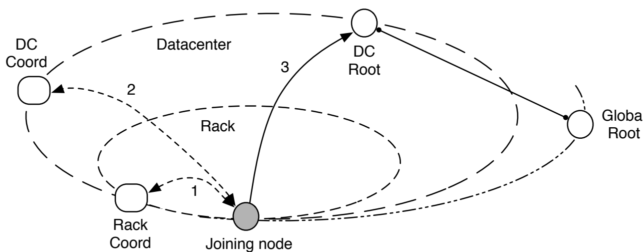

Fig. 3. DC2-Multicast tree construction using cluster coordinators to identify roots

and bounded. Running the RandTree protocol across *all* the nodes in the network will involve large overheads in join, as for every node that joins the tree, the join messages need to propagate from the root all the way to a child that has capacity to accept this node. What makes RandTree even more attractive as a design choice for DC2-Multicast is that, in a datacenter environment, the cardinality of clusters is typically bounded. For instance, each rack contains a maximum of 50 machines, and the total number of racks in a datacenter is often limited to few hundred. These bounds make the construction of multicast trees using RandTree very efficient and compact.

The following procedure is used by a joining node to find its tree members as well as construct parts of the tree that do not yet exist. After the group key is computed, the cluster coordinator for the group key at the lowest hierarchy is identified. A cluster coordinator responds to a *RootRequest* for an unknown group with the address of the same node (forcing the first node to become the cluster root). All future requests for the same group would be replied with the identity of the cluster root. Following the root discovery phase, the root continues to build the tree at the upper hierarchy ($h-1$), whereas a non-root contacts the root to join the tree.

Fig. 3 illustrates the process applied by a joining node in the construction of DC2-Multicast trees for a two-level datacenter hierarchy. The joining node identifies the key associated with the lowest rack cluster by substituting all of its location identifiers into the group key (at $h = 2$). Further, it contacts this cluster coordinator (*Rack Coord*) with a *RootRequest* message as shown by arrow 1. In this example, the joining node happens to be the first node in this rack cluster for the group, and is nominated as the cluster root (*Rack Root*). The join process recursively continues at the joining node for the upper hierarchy ($h = 1$), and identifies the datacenter cluster by appropriate substitution in the group key. Arrow 2 shows the routing of the *RootRequest* to the cluster coordinator (here *DC Coord*). At this step, the *DC Coord* identifies the presence of an existing cluster root (marked *DC Root*), and returns the address of this node. The joining node proceeds to send a *Join* to the *DC Root* to join under the tree (arrow 3). Since a tree for this group already exists from the datacenter upward, the joining node has successfully become a part of the global tree.

Benefits: DC2-Multicast minimizes the inter-cluster tree edges to the minimum, and creates well balanced trees because only the tree root at each cluster participates in the tree construction at the higher cluster. One of the key advantages in DC2-Multicast is participation of only the subscribers in the tree for a group. Although we involve coordinator nodes picked at random, they are passively involved in the tree construction and do not contribute resources toward data transfer. This reduces the node stress for each group tree, and hence lesser processing is required globally during multicast. DC2-Multicast load balances the load on the coordinators for different groups, and maintains low state overhead on each tree member. Clustering helps to keep failures localized, and easily manage a small number of nodes at each hierarchy. Finally by keeping the node and network footprint of each multicast tree low, the overall load on the system is minimized. These properties indicate that DC2-Multicast is highly *scalable* to the order of 200K servers in today's datacenters.

The assignment of location identifiers could leave the global keyspace unbalanced, creating hotspots. But we argue that through good design of the application, the total traffic at the global hierarchy can be bounded by the number of data centers. Control messages exchanged for new joins, leaves or node churn always proceeds from the lowest hierarchy up to the highest. This keeps the control overheads in real conditions such as failures closer to the node, avoiding slow links.

C. Application 2: DC2-Store

DC2-Store is a caching Key-Value store application that allows geo-distributed applications to operate a distributed storage of objects. A common approach to reducing lookup latency in a DHT store is caching the object at the root's K successive nodes [14]. Such techniques primarily depend on the randomness in assignment of keys to nodes in a DHT, so that one of the replicas is nearby. DC2-Store is however geared toward providing stronger locality of caches, and thereby reducing lookup latency by finding a nearby cache. By virtue of the cluster hierarchies of DC2, caching is performed at different proximity levels between the original object location and a client.

DC2-Store uses a reactive caching strategy where nodes between the client to the object location cache the object on the reverse path. DC2 provides cluster coordinators that exist at increasing proximity from the client, with the global coordinator being the authoritative storage of the object. DC2-Store forces the forward path of the lookup to be routed through the cluster coordinators, and caching them at these nodes on the reverse path. Algorithm 1 captures the essential steps involved in processing the lookup for a key. If a key is found in the cache at any intermediate coordinator during the lookup, it is immediately returned without traversing further. The lookup reply similarly cascades its way down the cluster hierarchies, caching itself at coordinators. A store operation (not shown here) takes the standard approach in routing the value to the key owner and adding the pair to its store.

The design of DC2-Store minimizes client latency for object lookups, through intelligent replica choices for caching. A lookup will always traverse the minimum overlay distance required, and does not travel outside the smallest enclosing cluster holding a copy. *i.e.*, in a datacenter, if the datacenter

Algorithm 1 DC2-Store

Algorithm: Lookup
Input: Key key
Input: Location Identifiers $identifiers$
Input: Hierarchy hy
  $next_coord \leftarrow$ next coordinator from DC2 using key and $identifiers_{1..hy}$
  $\mathsf{Send}(next_coord, \mathsf{Lookup}(key, hy))$
  $value = \mathsf{Receive}(next_coord, \mathsf{LookupReply})$
  return $value$

Algorithm: Deliver
Input: Key key
Input: Sender s
Input: Hierarchy hy
  if $hy = 0$ **then** {Global coordinator/key owner}
    if $key \in store$ **then**
      $\mathsf{Send}(s, \mathsf{LookupReply}(key, store[key]))$
    end if
  else {Cluster coordinator}
    if $key \in cache$ **then**
      $\mathsf{Send}(s, \mathsf{LookupReply}(key, cache[key])$ {Reply from cache}
    else
      $value = \mathsf{Lookup}(key, hy)$ {Forward lookup}
      Insert $(key, value)$ in $cache$
      $\mathsf{Send}(s, \mathsf{LookupReply}(key, value))$
    end if
  end if

coordinator has already cached a key, future requests for the same key will not exit the datacenter network. Although the above description shows a reactive mechanism for caching, it is fairly straightforward to imagine a proactive replication scheme. Such a scheme would serve to reduce client latencies, and simultaneously place object replicas in physically separate nodes improving its fault-tolerance.

We outline the design of a store that leverages location in this data center setting. There has been a good amount of literature in techniques that provide fault-tolerant replication, cache coherency, caching policies, etc. which are orthogonal and beyond the scope of this work. But, the location information provided by DC2 can be used to enhance their potential. As an example, the fault tolerance of replication is strengthened by choosing replica locations that are spread out geographically, because simultaneous failures are usually colocated.

III. Implementation

Unmodified Scribe is layered on top of Bamboo to provide overlay multicast trees. Bamboo interacts with other peers directly using TCP/IP. DC2 is implemented as a service that encompasses the *Location Identifiers* and *Bamboo* modules to provide the functionality of overlay routing. It also exposes the `getCoord` method for identifying cluster coordinators to services layered above. DC2 nodes are given their location identifiers at startup based on their location in the experimental topology.

DC2-Multicast is implemented as a combination of DC2 for root selection and RandTree for tree construction, combined with additional sets of routines to handle failures. DC2-Multicast uses DC2 only during tree construction for finding the cluster roots, after which RandTree uses TCP/IP directly for communicating with parents and children in the tree and also during data transmission. The implementation of DC2-Multicast including the integrated RandTree construction protocol constitutes about 800 lines of C++ code in Mace [15] (a toolkit for building systems), and the implementation of DC2-Store requires about 300 lines of additional C++ code.

Both the multicast implementations–Scribe and DC2-Multicast, provide the same basic abstraction of *group trees*. Group tree abstraction allows the upper layer to specify group subscriptions based on a group key. The upper layers can query for the parent and children in the tree for the respective group by providing a group key. The multicast application that we use to test the multicast functionality and performance of the protocols is layered over the group tree abstraction. A node takes in a *subscription* file to identify the groups, for which it should issue `subscribe` requests during an experiment. The application also takes in a *schedule* file that identifies the communication involved in the groups. This schedule identifies a timestamp, along with the node that should multicast a message in a specified group. DC2-Store exposes the same abstractions as a DHT with `get` and `put` methods to insert and retrieve objects associated with a key. The key is constructed as a SHA-160 hash of the key string.

IV. Evaluation

This section describes our experimental setup, metrics and a detailed discussion of the experiments on two DC2 applications. We demonstrate the benefits of DC2-Multicast through a performance comparison against Scribe. We also present a comparison of DC2-Store with Passive caching to further highlight advantages of topology-awareness in data centers.

A. Experimental setup

Our experiments were conducted over an emulated topology of data centers to facilitate a large network setup. Our testbed consists of 15 machines equipped with 8-core 2.33 GHz Intel Xeon processors and 8GB of RAM running Gentoo Linux 2.6.32.3. All the 15 machines are connected through a switch using 3 Gbps links (3 x 1Gbps interfaces bundled together).

We used Modelnet [16] to emulate a hierarchical network topology (similar to a typical data center, except with fewer number of levels in the hierarchy) as shown in Fig. 4. This emulated topology consists of two data centers with seven racks each, each rack consisting of 50 machines, giving a total of 350 virtual nodes per data center, and 700 nodes overall. Each node is executed as a process in the testbed with up to 50 on a single machine. Modelnet [16] emulates virtual topologies inside a single machine, by intercepting

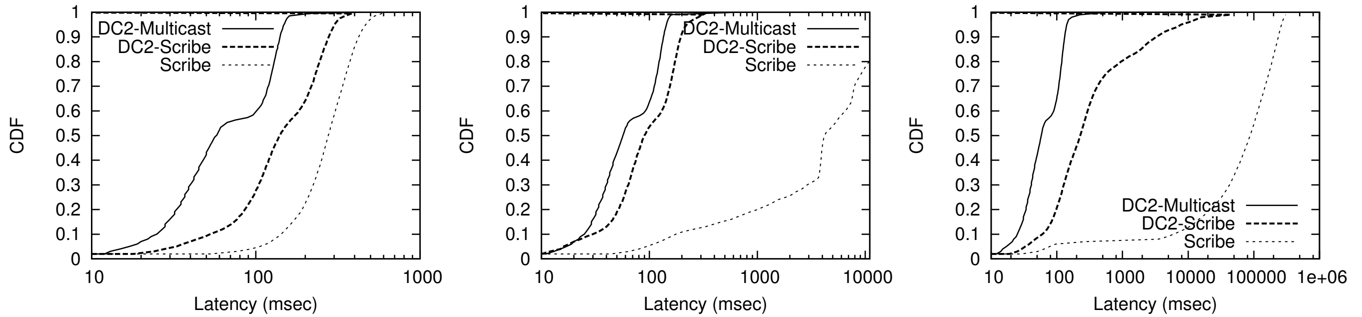

Fig. 5. CDF of latency for packet injection rates of 2, 10, 50 per sec

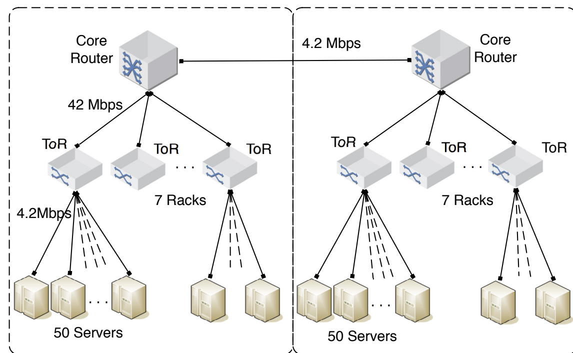

Fig. 4. Hierarchical Data center Topology with 7 racks each in 2 data centers

traffic at each machine and simulating the virtual latencies and queuing delays. Given the physical bandwidth limitation for the Modelnet core emulator, each of the 700 virtual nodes cannot be guaranteed more than 4.2 Mbps bandwidth.

All the racks have a Top of Rack (ToR) switch connected to hosts using 4.2 Mbps links with 1 ms latency. The ToR switches are all connected to the core router for the data center on 42 Mbps links with 2 ms latency. The core routers are inter-connected by a single 4.2 Mbps, 50 ms link. The link bandwidths are assigned to follow the trend which is similar to typical 'scale-up' data center topologies, i.e., the link bandwidth of the ToR-Core router link is about 10× more than the end host link. We assume that the inter-data-center link carries the same bandwidth as the end host link, although with much higher latency (of 50 ms) typical of WAN links.

B. Multicast performance

Our evaluation of DC2-Multicast compares performance with Scribe as the baseline, because to our knowledge only Scribe currently scales well to large group sizes, and dynamic group memberships across many groups in a decentralized manner. Note that Scribe leverages the location of its nodes through latency measurements by the Pastry service. IP Multicast does not scale well across data center boundaries, and hence we do not consider it in our evaluation. To facilitate a thorough evaluation, we designed and implemented an intermediate solution called DC2-Scribe that is based on Scribe. DC2-Scribe diverges from Scribe during a node join, by using an overlay route that is forced through the DC2 coordinators at each hierarchy, thus clustering routes based on location.

At the beginning of each experiment, all the virtual nodes are brought up on the 14 machines and given sufficient time to stabilize the overlay ring. We use 700 groups of varying size with the group sizes following a Zipf-ian distribution. Indirectly, this also implies that a number of subscriptions by a single node follows a Zipf distribution. These group sizes are reflective of real world applications such as social networks, with variable group sizes (following a heavy tail distribution) and group memberships.

The minimum, maximum and average group sizes are set as 25, 150 and 75 respectively, and an exponent value of 3. Nodes are randomly assigned to groups, with a single node assigned to be the multicast source for the group. All of the systems under consideration support multiple sources per group, and the use of a single source is only for experimental purposes. For these experiments, the group memberships remain constant throughout the experiment. The rate of packet generation is globally uniform for the system with a fixed number of multicasts each second. Random nodes are picked to initiate multicast on their assigned groups using 1KB packets. We have chosen these numbers somewhat arbitrarily for illustration purposes; real benefits of our system may to some extent depend on the particular deployment scenario.

1) Latency: Fig. 5 shows the CDF of all message latencies (x-axis in log scale). We show the CDFs for packet rates of 2, 10 and 50 pkts/sec. Note that the CDF is across all nodes and all multicast groups. At 2 pkts/sec, the median latency for Scribe is 277ms, DC2-Scribe is 141ms and DC2-Multicast is only 58ms, showing significant latency boosts. At 50 pkts/sec, Scribe breaks down and is able to deliver packets with a median of 87 seconds, whereas the median for both DC2-Multicast and DC2-Scribe remains the same. Latency is the one of the most popular application perceived metrics.

We observe that when the groups are lightly loaded, at 2 pkts/sec, the difference between DC2-Multicast and Scribe is the least, compared to heavy loads at 50 pkts/second. As we increase the packet injection rates, the latency experienced by nodes in Scribe is much higher compared to either DC2-Scribe or DC2-Multicast. The improvement exhibited by both

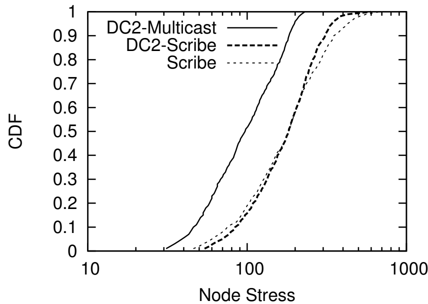

Fig. 6. CDF of Node Stress

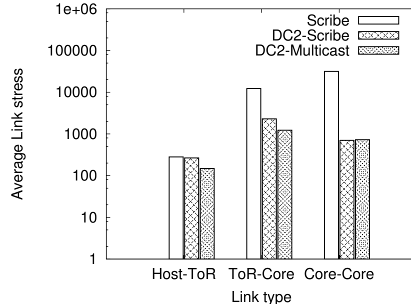

Fig. 7. Categorical Link Stress on physical links

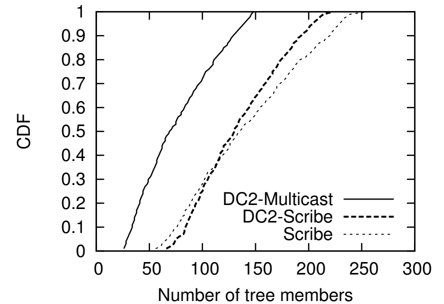

Fig. 8. Number of Subscribers

DC2-Multicast and DC2-Scribe can mainly be attributed to the reduced use of the inter-data-center link. This link alone contributes 50 ms of latency and frequently congested; thus avoiding this link as much as possible gives better performance. The bandwidth limit on this link actually results in the breakdown of Scribe's performance at high traffic rates. Since the trees constructed by Scribe are not localized to within data centers, there could be multiple tree edges along a single root-to-leaf path which make use of the inter-data-center link. DC2-Multicast performs even better than DC2-Scribe, because tree construction over overlay paths in DC2-Scribe involves non-subscriber intermediate nodes, leading to skinnier trees.

2) Node and Link stress: Low latency is nice, but it is also important to understand how DC2-Multicast affects *node-* and *link-stress* as they directly relate to congestion on the nodes, and individual link usage. We define node stress as the total number of messages that a node receives in any of the trees it participates. Using the same experiments as above at 10 pkts/sec, Fig. 6 plots a CDF of node stresses on all 700 nodes (note that the x-axis in log scale). The median node stress on the Scribe flavored nodes is 182 and DC2-Multicast has a stress of 97. DC2-Multicast trails roughly at about half the node stress from Scribe, while the maximum is almost $3\times$ lesser for DC2-Multicast. Node stress directly translates to the number of packets that is handled by a node during multicast, and hence lesser node stress implies lesser load. It also implies that the local network bandwidth usage on the node is minimized. The reason behind lesser node stress is the property of DC2-Multicast to involve just the subscribers of the group in the tree. Thus any node would only receive messages (and hence forward) only on the trees it is subscribed. It should be noted that the actual number of nodes in a DC2-Multicast tree equals the number of subscribers.

DC2-Scribe performs similar to Scribe because of the similar mechanism used to construct tree edges, by involving intermediate nodes in the path to root. On closer observation, Scribe seems to have a higher percentage of nodes exhibiting lower amounts of stress than DC2-Scribe, but the curves cross each other at around the median. At larger values, the trend reverses and stress on Scribe nodes is larger. We noticed, however, that on an average, nodes in Scribe exhibit larger stress. We repeated this experiment with a larger average group subscription, and the results were in similar spirit with DC2-Multicast consistently keeping lower node stress.

Link stress: Link stress is defined as the total number of messages sent on a link for a single multicast. We measure link stress by multicasting a single message on all groups, and calculating the number of messages carried over each link. Using the source and destination of these packets and the corresponding unique Modelnet route, we can associate packets to physical links. We considered link stress because it indirectly relates to the bandwidth consumption on the links. We have 3 types of links in this topology: *Host-ToR* (end host and ToR switch), *ToR-Core* (ToR switch and core router), *Core-Core* (inter-data-center link between core routers). The link stresses are plot as averages categorized based on the link type in Fig. 7 (note that the y-axis is in log scale).

The links are shown in increasing order of cost, traffic and hence importance of optimizations. The link stress for the *Host-ToR* link in DC2-Multicast was 148, which is about twice the size of the average group in this experiment, while those of the Scribe varieties is about 270. DC2-Multicast shows link stress values that are close to ideal for overlay multicast techniques, primarily because of involving only subscribers in tree construction. The differences widen for costlier links and the hierarchical techniques have almost a magnitude of lesser link stress. The most prominent in this figure is the performance gains of over two orders of magnitude from DC2-Multicast. The link stress on the *Core-Core* link was about 700 for DC2-Multicast, which matches the number of groups in the system. Given the random assignment of nodes to groups, each group has subscribers in both data centers and hence the link stress is equal to the ideal. The DC2-Scribe implementation deviates by a factor of 2 from the ideal (DC2-Multicast) because of some of its persistent issues from Scribe, including taller, skinnier trees.

We also experimented link stress for group subscriptions with larger number of nodes, and the gap between the systems grows wider. DC2-Multicast continues to maintain close-to-ideal link stress values. Link stress for inter-data-center links directly translates to network cost, and by optimizing on the location, DC2-Multicast saves operating costs by multiple orders of magnitude. Node and Link stress are direct influences in the scalability of the system, in terms of reducing the

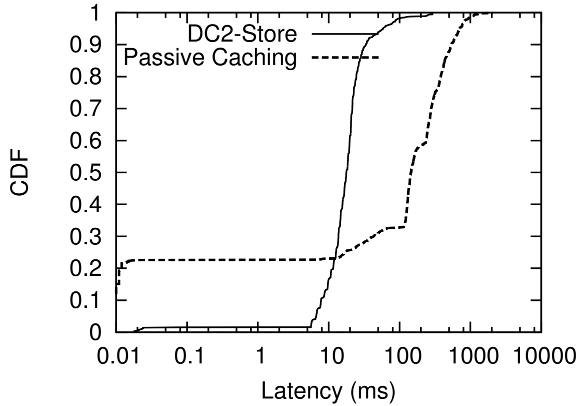

Fig. 9. CDF of Lookup latencies in a Caching DHT

total overhead on the individual nodes and the network. These experiments show the potential for scalability and cost-effectiveness of DC2-Multicast.

Number of tree members: In order to expose the actual number of nodes involved in tree construction, we plot the CDF of the tree size for each group in Fig. 8. By construction, DC2-Multicast has equal number of subscribers and tree members, and its curve is ideal. Both Scribe and its DC2-Scribe have almost 2 $\times$ more tree members than subscribers. Similar to the node stress figures earlier, Scribe and DC2-Scribe cross over each other. An interesting observation is that Scribe trees have lesser members for smaller groups. This is explained by the forced detours that DC2-Scribe is subject to, before reaching the root, deviating slightly from the shortest path considered by Scribe. As group sizes grow larger, the height of either trees (Scribe and DC2-Scribe) increases, and also the overall node stress. Increasing the height of the tree would mean larger latencies for leaf nodes. In keeping the number of nodes in the tree to a minimum, DC2-Multicast aims at solving these problems and construct trees of lesser height, decreased node stress and decreased end-to-end latencies.

C. Key-Value store

We compare DC2-Store against Passive caching which is a technique proposed in [14]. In this technique, an object is stored on all nodes on the reverse query path to the destination. Both DC2-Store and Passive caching are perform reactive caching upon lookups for objects. Every 1 second, each of the nodes picks a random key on a Zipf-ian scale out of a total of 50 keys, and performs the lookup for this key (to simulate popular objects).

Fig. 9 shows a CDF plot of the lookup latencies experienced by all the nodes during a run (x-axis in log scale). The median latencies for passive caching was about 144ms while that of DC2-Store was only 17ms which is an order of magnitude of faster accesses. It is interesting to see that about 20% of the latencies for Passive caching is lesser than those of DC2-Store, because every single node on the query path caches objects. However, DC2-Store only caches at 2 distinct nodes in the query path to root, and the total number of cache sites is limited. DC2-Store benefits from a large chunk of the latencies by an intelligent choice of the query path and locality-aware caches. A DHT lookup that is constructed using location awareness can hugely benefit from site-local caches, and hence scale to better throughput for geographically distributed nodes.

V. RELATED WORK

Location-awareness has been adapted to overlay group communication techniques [5], [11], [17], but this is achieved using latency estimations, and is only a secondary criteria in optimizing overlay paths. Latency is not a good estimator of the physical location of nodes within datacenters because of close proximity, traffic changes and virtualization [18]. To the best of our knowledge, our work is the first in tailoring the construction of overlays for group communication to datacenter environments. Hierarchical overlay routing [19], [20] has been proposed for constructing single and multiple rings for each hierarchy, but the node overhead is large for many hierarchies. DC2 has a small overhead by maintaining a single routing table per node, and provides location properties easily usable for both multicast and Key-Value store applications.

NICE [11] proposes a hierarchical overlay construction protocol by clustering together nodes with lower end-to-end latency. A source specific tree is constructed on top of the overlay for multicast. However, it does not provide a scalable mechanism for constructing *multiple* independent trees. Also, it depends on latency measurements on the network, which is susceptible to noise [18] and hence lead to non-optimal trees inside datacenters. Census [21] provides location-aware membership management and multicast and also providing solutions to Byzantine faults, but it is again limited to a single group unlike multiple dynamic groups supported by DC2-Multicast. In the context of today's social networks and e-commerce applications, large dynamic groups are a requirement to support client functionality.

AMMO [10] describes a mechanism to optimize constructed trees simultaneously for multiple parameters in a scalable manner. It is possible to use AMMO for selective optimization of certain clusters of DC2-Multicast, but this is orthogonal to the actual tree construction. We argue that specifying optimization metrics that optimize for the complete tree in data centers is hard, and do not guarantee link stress minimization. There is also an unnecessary cost of continually trying to optimize the tree safely. Our mechanism attempts to scale to large number of trees by minimizing the overhead in maintaining trees.

Kademlia [22] uses the XOR metric to construct routing tables and route to a given key. In our current implementation with Bamboo, it is possible that the search for a coordinator end at a node outside the cluster (because the key is closer to a node in the next cluster). The use of a XOR metric for key distance would solve this problem and avoid cross-hierarchy traffic even in corner cases.

Internet indirection infrastructure [23] aims at decoupling senders and receivers on the Internet by having an intermediate mapping. It uses Chord to consistently identify a rendezvous node which maintains the mapping of all receivers interested

in some sender's messages. This concept is similar to Scribe and our own mechanisms, but it does not explicitly provide location-aware services. SAAR [24] separates the control plan from the data plane of building a multicast overlay. This is to utilize shared overheads of control maintenance and probing across multiple groups and overlays. This is orthogonal to the DC2 partitioning of the graph into hierarchical units—SAAR could be used if needed to construct overlays within a unit.

Dr. Multicast [25] focuses on overcoming scalability limitations of IP multicast, by merging similar groups and devising a hybrid IP multicast and IP unicast solution to provide multicast service *within* data centers. However, this will only be effective when IP multicast is enabled in the platform. DC2-Multicast cannot match the efficiency of IP multicast; however, by focusing on minimizing the link stress of expensive links, we do achieve many of the benefits of IP multicast, without inheriting its limitations. Also, there are no solutions to applying IP multicast across multiple data centers, which is essential for our targeted applications.

VI. Conclusion

We argue in this paper that group communication systems (*e.g.*, multicast, DHTs) form a key building block for several applications ranging from social networks to e-commerce and bidding, and need to be revisited in the context of today's datacenters. Our motivation stems from the emergence of hierarchical datacenters and cloud environments, where links tend to be quite heterogeneous in nature. Further, the operators of these services in these environments have incentive to avoid costly aggregation or core links in a typical scale-up datacenter architecture which currently run with large oversubscription ratios. A similar argument applies in the context of avoiding inter-data-center links in cloud environments as much as possible to reduce the cost incurred in running such large-scale services.

We present a simple group communication framework called DC2 that allows nodes to be aware of their location within the hierarchy, and minimize cross-hierarchy communication as much as possible. We build two systems—DC2-Multicast and DC2-Store—on top of DC2 to demonstrate the benefits of DC2. In our evaluation using a prototype implementation, we observed that DC2-Multicast results in 2-3 $\times$ reduction in the node and link stresses, and reduces median latency by several orders of magnitude compared to Scribe. Similarly, our experiments show that DC2-Store achieves lookup latencies lower than the default by a factor of 8. While the results may be limited by our experimental resources and exact benefits may vary, we believe that the core ideas embodied in DC2 will prove to be useful in redesigning many different group communication systems, that can now be revisited in the context of hierarchical datacenters and global cloud environments.

Acknowledgment

This research was supported in part by the National Science Foundation (NSF) under grants 1054567-CNS, CNS-1054788 and IIS-1017898. Any opinions, findings and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] "Facebook statistics," http://www.facebook.com/press/info.php?statistics.
[2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A Large-scale and Decentralized Application-level Multicast Infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, 2002.
[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of SIGCOMM*, 2001.
[4] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Proceedings of Middleware*, 2001.
[5] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling Churn in a DHT," in *Proceedings of USENIX ATC*, 2004.
[6] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: A Public DHT Service and its Uses," in *Proceedings of SIGCOMM*, 2005.
[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of SOSP*, 2007.
[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proceedings of SIGCOMM*, 2009.
[9] "Amazon Web Services," http://aws.amazon.com/.
[10] A. Rodriguez, D. Kostic, and A. Vahdat, "Scalability in Adaptive Multi-Metric Overlays," in *Proceedings of ICDCS*, 2004.
[11] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable Application Layer Multicast," in *Proceedings of SIGCOMM*, 2002.
[12] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination using an Overlay Mesh," in *Proceedings of SOSP*, 2003.
[13] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat, "Using Random Subsets to Build Scalable Network Services," in *Proceedings of USITS*, 2003.
[14] A. Rowstron and P. Druschel, "Storage management and caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility," *SIGOPS OSR*, vol. 35, no. 5, pp. 188–201, 2001.
[15] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language Support for Building Distributed Systems," in *Proceedings of PLDI*, 2007.
[16] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and Accuracy in a Large-scale Network Emulator," in *Proceedings of OSDI*, 2002.
[17] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr., "Overcast: Reliable Multicasting with an Overlay Network," in *Proceedings of OSDI*, 2000.
[18] G. Wang and T. S. E. Ng, "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," in *Proceedings of INFOCOM*, 2010.
[19] P. Ganesan, K. Gummadi, and H. Garcia-Molina, "Canon in g major: Designing dhts with hierarchical structure," in *Proceedings of ICDCS*, 2004.
[20] M. S. Artigas, P. G. Lopez, J. P. Ahullo, and A. F. G. Skarmeta, "Cyclone: A Novel Design Schema for Hierarchical DHTs," in *Proceedings of IEEE P2P*, 2005.
[21] J. Cowling, D. R. K. Ports, B. Liskov, R. A. Popa, and A. Gaikwad, "Census: location-aware membership management for large-scale distributed systems," in *Proceedings of USENIX ATC*, 2009.
[22] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Proceedings of IPTPS*, 2002.
[23] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," *SIGCOMM*, 2002.
[24] A. Nandi, A. Ganjam, P. Druschel, T. S. E. Ng, I. Stoica, H. Zhang, and B. Bhattacharjee, "SAAR: A Shared Control Plane for Overlay ," in *NSDI*, 2007.
[25] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock, "Dr. Multicast: Rx for Data Center Communication Scalability," in *Proceedings of EuroSys*, 2010.