P2P Objects

Wenyan Dong, Sonam Dorji, Sachin Katti and Dimitris Vyzovitis

ABSTRACT

Massively distributed systems like the Internet present new challenges for scalable distributed querying due to their sheer size. Hence there is a need for new techniques to organize information and query them in a scalable fashion.

We present a novel method of organizing information based on content based multicast groups which mimic a hierarchical naming scheme. The hierarchical naming enables efficient partitioning of different types of data. The corresponding structure in the multicast groups enables quick location of a subset of nodes containing certain object types. Quick location of the rendezvous point (RP) of a multicast group is obtained by storing the group names and the corresponding RP in a DHT. More sophisticated message passing primitives including *k*-cast and anycast are also provided so that applications can choose the appropriate method depending on their consistency requirements.

We evaluate our algorithms using *p2psim* [2] and show that they provide good performance when compared to the ideal multicast trees which they approximate at worst by a factor of 3. The group formation and maintenance cost is also shown to stay constant after a certain size. Finally we provide an implementation on top of the Bamboo DHT system provided by the OpenHash [1] project.

1 OVERVIEW

Internet scale querying systems are hard to build and maintain because of the large number of nodes. As the number of nodes grows, it becomes exceedingly difficult to organize information and query them in a scalable fashion. At present, the closest example to such a distributed querying system is file sharing and the search mechanisms built into current client implementations. A precursor to present day P2P networks was Napster. Napster accumulated all meta-data in a central server, which acted as a query server. Present day networks such as Gnutella spread the responsibility and run truly distributed queries over a large scale graph. This stems from the need to spread responsibility and avoid single points of failure. These networks essentially do random walks on the graph and collect information about nodes which might satisfy the query. The random walk technique results in good response times for popular files but very bad worst case performance. In addition, these queries presently are limited to keyword searches based on file names. We can easily envision a future in which users might want more rich searches based on other file meta-data. An important point to note here is that the data is still with the users and is not placed some place else for enforcing some structure. Hence these queries are run on unstructured distributed systems.

Such large scale distributed querying has many interesting applications. Consider a deployment of sensors over a large scale geographic area. There are subsets of sensors which talk to a central node connected to the Internet. Querying such a system is much more complex than keyword style file searches. The essential feature of this type of system is the live data which is interesting for a fixed time scale, making it impractical to store it in a central server since it would require frequent updates. Also queries need to be accurate, i.e. if a query is for all nodes having seen a particular attribute (to quote the classic example an elephant passing by), all such nodes need to reply. Hence there are stronger consistency requirements on these systems.

Search mechanisms which use random walks are unsuitable since they return incomplete results. They also do not scale as the system grows larger. Hence there is a need for more scalable, structured querying. But Gnutella style overlays are ill-suited for this. They offer great flexibility in the way the overlays are formed (essentially the graph is random) but the penalty is paid in the cost for queries.

DHTs (Distributed Hash Tables) have been evolved with the above concern in mind. DHTs are structured overlays in which data is stored in specific nodes depending on what value they hash to. Due to the rigid structure on data placement, exact lookups are very fast. State of the art DHT implementations require only $O(logn)$ hops to locate the node which holds the data. This is achieved by a binary descent on the hash key space. Unfortunately DHTs are promising only for exact lookups where the querying agent already knows the exact key of the data he is looking for. In the querying systems we mentioned above, nodes want to do a distributed query on a large number of nodes without knowing the exact keys. DHTs presently are ill-suited for this.

We present new methods for building scalable distributed querying systems based on content based groups. The basic intuition is to use a hierarchical namespace and build content based groups at various levels of the namespace. Hierarchical namespaces give us a way to organize information more effectively and also narrow down the querying set considerably. Building groups at various levels of the namespace, we can then restrict our queries only to the tree corresponding to that hierarchical level. This eliminates the uncontrolled flooding presently employed. Nodes join different groups based on the type of objects they hold. For example in a file sharing system we would have groups for artists and queries for songs of a particular artist would get multicast only on the group corresponding to that artist. Similarly for the sensors example above, a query for elephant sightings would be multicast to the group corresponding to sensor networks which monitor elephants and so on. As the query propagates through the multicast tree and the replies start accumulating, the replies travel in the reverse direction to the query and the root of the tree replies to the querying node.

Some applications might not need the strong consistency provided above, i.e. they might not care about getting replies from all nodes satisfying the query in the group. An obvious ex-

ample is service discovery of any kind, in which the application needs to know of only one instance of the service. This is easily done in our content based groups by implementing an anycast primitive for sending query messages. Nodes will then receive a reply from the nearest node which can satisfy the query. The primitive can be generalized to do a *k*-cast where *k* is the cardinality of the replies which the querying node expects and cares about.

In this paper we specifically describe algorithms for building efficient content based multicast trees. Previous techniques to do this build the tree using the underlying DHT routing. Message routing was based on a hash of the name of the multicast group. Nodes wishing to join such a group would send a message to the nodes holding the hash key of the group name. All intermediate nodes on the path of this message would also join the multicast group. Hence the whole tree is not optimized for network distances.

We take a different approach. Instead of using the DHT routing for building the trees, we use the DHT essentially as a data structure for holding information about rendezvous points. Hence the node responsible for the hash key of the group name contains the pointer to the root of the tree. Similarly as we go down the hierarchical namespace, the nodes responsible for the name will hold the information about the root of the tree at that level.

Trees are optimized to minimize the total weight of the edges in the tree. The weight of the tree is presently considered to be the network latency, but it is easily extended to optimize based on other parameters such as network bandwidth etc. Nodes when joining a group select the most appropriate place to join based on which existing node is closest to them. Upon node failures and at periodic intervals a stabilization algorithm can be run in order to balance the tree and maintain tree efficiency.

In the following sections we elucidate further on these ideas. Specifically, we discuss application design in Section 2. We elaborate on the design and describe the core algorithms of the system in Section 3, while we provide performance evaluation in Section 4. Finally, Section 5 discusses related work in the area and provides background on the P2P Objects design, and Section 6 concludes the paper.

2 APPLICATIONS

Before proceeding further we motivate the usefulness of our primitives by outlining how a typical distributed application would use our communication primitives to implement scalable querying. The application space we consider is that of instances running on semi-reliable end hosts producing live semi-transient data. We envisage distributed large scale queries being served by this system.

2.1 Hierarchical Naming

In the distributed system we envisage, if data is allowed to be arbitrarily named and be part of random groups, efficient location of the part of the tree which might hold information about particular data will be very difficult. For example if song files of a particular artist are allowed to be placed in arbitrarily random places in the tree, it is very tough to locate the subset of nodes where we could find such files in a scalable manner. Hence we propose a hierarchical naming scheme for objects to enable scalable discovery. Each node which produces data, names it in an appropriate manner according to the hierarchical naming scheme. We describe the scheme with an example.

Consider a music file which could be an object in a content distribution P2P network. The file can be hierarchically named to belong to a particular set of groups. Here is an example:

```
cnt
  music
    artist
      aphex-twin
         classics
           digeridoo
```

In this example, ”/cnt/music/artist/aphex-twin/classics/digeridoo” represents a data object which contains the file for that track, ”/cnt/music/artist/aphex-twin/classics” represents a directory object that contains all tracks in Aphex Twin’s ”Classics” album, and so on.

This naming mechanism can be generalized to objects of all types; not just music files. Once we have such a naming mechanism in place, it is easier to index objects belonging to a particular object type. Nodes just join (or form a multicast tree if none exists) for that object type. For example, consider an object accessible through /a/b/c. A node instantiating an object /a/b/c is part of the /a, /a/b, and /a/b/c spanning trees. The path of an object maps to an entry in the DHT which contains the root of the respective spanning tree. For example the nodes responsible for the hashkeys of /a, /a/b and /a/b/c would contain pointers to the IP addresses of the nodes which are roots of the corresponding subtree. Each node in the tree maintains pointers to its parent and children. The tree is maintained in a distributed fashion as described later.

A note on how nodes join the appropriate place is in order here. When a new object is inserted into the system, for e.g. /a/b/c, the node responsible must insert itself into the tree corresponding to /a/b. If /a/b/c is itself an object type, it must either create a tree for /a/b/c type or join one if it exists. Hence it does a lookup on the DHT for the key obtained by hashing /a/b/c to check if a root exists. If it does it joins the tree, else it now goes up one level and looks up the root for the tree corresponding to /a/b/ and so on until it finds a tree to join or creates one itself. If it ends up creating the tree corresponding to /a/b/c then it inserts its own IP address in the DHT with the key being the hash of /a/b/c. Future nodes joining can therefore know that /a/b/c already exists and take appropriate steps.

2.2 Querying

Querying follows directly from the tree formation mechanism described below. Nodes wishing to query for a particular data object will multicast a query on the tree corresponding to that object type. They can do so by obtaining information about the rendezvous point from the DHT and then sending a message to the root which then multicast it down the tree. As each node receives a query message from its parent, it queries its own local database and sends its reply to its parent. As part of future

work we intend to investigate efficient aggregation mechanisms for the replies which travel up towards the root. If the querying node doesn't care about querying the whole group and receiving complete replies it can do a *k*-cast of the query message on the group. It will then only receive a reply from the root consisting only of *k* nodes which satisfy the query.

We illustrate with the above mentioned music file sharing application as an example. An application that requires to find what Aphex Twin albums are accessible in the system can use the "/cnt/music/artist/aphex-twin" path to multicast a "list" message to the tree corresponding to this path. The result will be a set of replies, containing the paths of all objects accessible in the system. In order to retrieve a specific file, say "/cnt/music/artist/aphex-twin/classics/digeridoo", the application can then anycast a "get-data" message. The result will be the file delivered by a node in close proximity in the tree.

There are several examples of applications which would use such a querying system. As we mentioned before a geographically distributed sensor network was one. Another example is distributed network monitoring in which the querying facility is built on top of standard network tools such as `tcpdump`. Essentially any application which produces rich live data and needs rich predicate based querying can employ the P2PObjects primitives.

3 DESIGN

In this section we describe the design of the P2P Objects core algorithms for building, maintaining and routing in multicast trees. The basic address scheme is the *group*, which describes an object type. Each tree is described by a group identifier, and all members of the tree are members of the group, i.e. implement the specific object type.

In order to send a message of a given cardinality, a node sends a message to the group. As messages propagate along the tree, nodes decide whether to accept or reject the message individually. Hence, a *multicast* or a $k - cast$ can be performed in a simple yet efficient way. Generalized queries and predication are supported by allowing the nodes to decide whether to accept or reject the messages.

In order to build and maintain group trees, we use an underlying DHT as a rendezvous point. DHTs provide a useful abstraction for a globally consistent data structure. We provide some background in DHT design and the interface required to support P2P Object trees in the sequel.

3.1 DHT

Distributed Hash Tables (DHTs) can be used to distribute and retrieve data among many nodes in a network. As the name implies, a DHT provides a hash table abstraction over multiple distributed nodes. Each node in a DHT can store a data item each of which is identified with a unique key. At the core of the DHT is a routing algorithm which delivers requests for a given key to the node responsible for that key. This is done without any global knowledge of the mapping of keys to nodes. The mapping could also be changing as nodes enter and leave the system. Routing proceeds in a multi-hop fashion: each node only maintains a small set of neighbors, and routes messages to the neighbor that are in some sense nearest to the correct destination.

DHTs provide strong theoretical bounds on the number of hopes required to route a key request to the correct destination, and the number of maintenance messages required to manage the arrival and departure of nodes from the network. There are many flavors of DHTs [12, 8, 14] which implement the above functionality. We specifically examine Chord [12] in the following paragraphs.

Chord's routing mechanism is based on a circle or a one dimensional logical space. Each node in the network is assigned to a point on the circle, also known as its *id*, and is responsible for any keys that map to the portion of the circle before its *id* and after the previous nodes *id* This is usually visualized as the arc going counter-clockwise around the circle. Routing can be achieved as long as each node knows its predecessor and successor on the circle. For efficiency each node also maintains pointers (known as fingers) to nodes throughout the circle. A finger is maintained for the node responsible for $id + 2^i$, for each i such that $0 < i < m$ and m is the number of bits in the identifier space.

A message is routed through the system in recursive style. A node will contact the node with the highest preceding value, compared to the target location identifier, within its finger table. That node will either accept the message, if it is responsible for the target location identifier, or forward the message with its best guess of the predecessor based on its finger table. The process repeats until the proper successor is located. On average a message requires $log(n)$ hops in the overlay network.

Tapestry [14] and Pastry [8] work in a similar way, by successively approximating the target node in their address space and operate in close theoretical performance bounds.

3.2 DHT as Rendezvous Point

The DHT serves an important function in the P2P Objects design: it stores the root node for each tree and provides a synchronization point. In order to join a group, a node locates a root in the DHT, and joins through this node. If no root exists for a given tree, then the node assumes ownership of the root.

A complication arises when dealing with concurrent operations of the tree. The usual DHT *put*/*get* interface provides no means for synchronization. A node assuming ownership of the root might be overridden by another concurrently joining node. Hence, nodes need to synchronize their operations without knowing about each other's intention. A naive solution to this problem is to use an external locking protocol; this solution however is expensive and unnecessary complicated. Instead, we provide an extended *put*! abstraction on top of the DHT, which provides synchronization in restartable atomic sequence fashion [4]. *put*! only uses the *lookup* operation provided by the DHT. In order to perform a *put*!, a node performs a *lookup* and supplies the old value associated with the key. If the value does not match the current value, because some other node has performed a *put*! in the meantime, the operation fails. In order to reduce communication overhead, a *put*! returns the current value, so that the faulting node can immediately join with the current root. The extended DHT interface and implementation

of the primitives used by P2P Objects is described in Algorithm 1

Algorithm 1 Extended DHT primitives

```
DHT.put!( k: key, v: value, ov:value ):
    n = lookup( k )
    if n == self:
        return self._put!( k, v, ov )
    else:
        return n.put!( k, v, ov )
DHT.put( k: key, v: value ):
    return self.put!( k, v, nil )
DHT.erase( k: key, ov: value ):
    return self.put!( k, nil, ov )
DHT._put!( k: key, v: value, ov:value ):
    cv = map[k]
    if cv == ov or cv == nil:
        map[k] = v
        return (#t, v)
    else:
        return (#f, cv)
```

3.3 Communication Primitives

For each group, we construct and maintain a spanning tree that can be used for routing messages. The spanning tree is maintained in a distributed fashion by a router service provided at each node. We describe the actual tree formation and maintenance algorithms in the subsequent section. Here we describe how applications interact with the router through the node interface assuming a group already exists:

```
interface node:
    // Application interface
    join( id: groupid )
    leave( id: groupid )
    send-msg( id: groupid, c: int, m: message ): int
    // Inter-Node interaction interface
    send( id: groupid, c: int, m: message ): int
    receive( id: groupid, c: int, m: message ): int
    insert-node( gid: groupid, nid: nodeid ):
        tuple<active: bool,
              parents: list<node>,
              roots: list<node>>
    erase-node( gid: groupid, n: node )
    set-parent( gid: groupid, n: node )
```

Applications use the *join* and *leave* primitives for editing membership to a group, and the *send−msg* method to send messages. Applications provide an upcall interface for the router to deliver messages:

```
interface application:
    deliver( id: groupid, m: message ): bool
```

3.3.1 Sending And Receiving Messages

Each node maintains a few data structures that describe the tree state for each group:

```
type group-entry: tuple<active: bool, parent: node, children:
list<node>, cache: circular-buffer<node>>
groups: map<groupid, group-entry>
apps: map<groupid, list<application>>
```

The children list for each group is maintained sorted by distance, which can be computed by a function d. d can be implemented by sending probe messages to the target node and caching the value for some particular time. Cached ping times can also be updated as messages are routed in the network. A sophisticated implementation can use network coordinate systems like Vivaldi [7] or an in-kernel implementation can access the *RTT* measurements from open TCP connections.

Based on these data structures, *send−msg*, *send* and *receive* are described in Algorithm 2. The core of the algorithm is *_receive*, which handles message delivery and routing, by recursively descending through a list of target nodes until the number of deliveries equals the cardinality of the message or the tree has been spanned. The list of target nodes is a sorted by distance list that includes the children and parent of the node, but excludes the source of the message. Multicast is handled as a special case, as there is no need to collect the number of deliveries; rather, the message is routed to all nodes.

Algorithm 2 Message Routing

```
node.send-msg( id: groupid, c: int, m: message ):
    if id in groups:
        return self.send( id, c, m )
    else:
        root = DHT.get( id )
        return root.send( id, int, m )
node.send( id: groupid, c: int, m: message ):
    return self._receive( id, c, m, #f )
node.receive( id: groupid, c: int, m: message ):
    return self._receive( id, c, m, #t )
node._receive( id: groupid, c: int, m: message, dd: bool ):
    delivered = #f
    if dd and groups[id].active:
      for a in apps[id]:
          delivered = a.deliver( id, m ) or delivered
    nodes = groups[id].children + groups[id].parent - sender-
of( m )
    if c == MULTICAST:
        for n in nodes:
            n.receive( id, c, m )
        return c
    else:
        rem = c
        count = 0
        if delivered:
            rem -= 1
            count += 1
        for n in nodes:
            if rem == 0:
                break
            count += n.receive( id, rem, m )
        return count
```

3.3.2 Tree Formation and Maintenance

Constructing and maintaining the group tree is a critical operation, as it determines communication efficiency and scalability of the system. We would like the tree to be efficient in terms of the latencies of the edges of the tree. In addition, we would like join and leave to be fast operations, so that the tree can be efficiently maintained in a dynamic environment. Ideally, we would like the tree to be the minimum spanning tree for the group. However, computing the minimum spanning tree requires global state and is difficult to maintain in a dynamic environment.

We construct the group tree as a C-ary tree with a proximity metric, with the cluster size C being the maximum number of children for each node. Each node maintains a pointer to its parent and each one of its children. The fundamental property of the tree is that the state is distributed, and *join*/*leave* operations do not affect the ability of the network to route messages.

3.3.3 Joining a Group

In order to join a group, a node looks up the root in the DHT. If the root does not exist, then the node becomes the root. If the root exists, then the node sends a $insert-node$ message to the root. The root returns the cached list of nodes that recently joined the subtree it maintains, which can be used as possible roots for the arriving node. If the root has less than C children, it includes itself in the list. In addition, the root provides with a list of its own immediate children, in order to deal with stale caches. If the root receives a second message from the same node and still has less than C children, then the arriving node becomes a child of this particular root. The arriving node sorts the list of possible parents by distance, and recurses until it successfully joins the tree, as illustrated in Figure 1. Note that the tree might become unbalanced in if distant nodes assume the root, hence a stabilization algorithm can repair the tree periodically (after a number of tree modification operations) or when inconsistencies are detected.

Formally, the implementation of *join* is described in Algorithm 3. For an example of how the algorithm works, consider Figure 1. In the figure, node n_1 wants to join a group rooted at *root*, with a cluster size 3. n_1 attempts to assume root of the tree in the DHT, an operation which fails returning the actual root. n_1 then proceeds with *_insert*, sending an $insert-node$ message to *root*, with nil reference. On receiving the message, *root* checks its data structure for the group, and determines that it has a full cluster. Hence, it returns to n_1 a copy of its cache consisting of $[n_2, n_3, n_4, n_5, n_6, n_7]$ and a list of its immediate children (not shown in the figure). n_1 sorts the cache by distance, determines that n_2 is the closest candidate root, and sends an $insert-node$ message to it. n_2 does not have a full cluster, and returns $[n_2, n_7]$ as candidate parents. n_1 sorts the new list, determines that n_2 is actually the closest node, and recurses to *_insert* on n_2, with a reference node n_2. This results to a new $insert-node$ message to n_2. n_2 still does not have a full cluster, and because the reference in the message matches its own identifier, it adopts n_1 as a child.

Algorithm 3 Joining a group

```
node.join( gid: groupid ):
    RAS:
    e = DHT.put( gid, self )
    if e.result:
        groups[gid] = group-entry( nil, [], [] )
    else:
        r = self._insert( gid, e.value, nil )
        if not r:
            goto RAS
node._insert( gid: groupid, root: nodeid, ref: nodeid ):
    r = root.insert-node( gid, self, ref )
    if not r.active:
        return #f
    if empty( r ):
        groups[id] = group-entry( root, [], [] )
        return #t
    sort( r.parents )
    for p in r.parents:
        if self._insert( gid, p, root ):
            return #t
    sort( r.alt )
    for p in r.alt:
        if self._insert( gid, p, root ):
            return #t
    return #f
node.insert-node( gid: groupid, nid: nodeid, ref: nodeid ):
    if not groups[gid].active:
        return (#f, [], [])
    if groups[gid].children.size < C and ref == self:
        groups[gid].children.add( node-entry( nid, d( nid ) ))
        return (#t, [], [])
    else:
        r = (#t, groups[gid].cache, [])
        r.alt = groups[gid].children - r.parents
        if groups[gid].children.size < C
            r.parents.insert( self )
        groups[gid].cache.insert( nid )
        return r
```

3.3.4 Leaving a Group

In order to leave a group, the node notifies its parent and provides the children with a new parent. The parent removes the node from the tree, while the children re-enter themselves using the new parent as the root. The subtrees of each child are not modified. If the leaving node is the root of tree, then it erases the DHT entry for the group and allows the children to content for assuming the root. The contention is resolved by synchronizing on the DHT with the remaining children joining the new root.

The implementation of *leave* is formally described in Algorithm 4, while Figure 2 provides an example. In the example, node n_1 decides to leave the group. In order to do so, it asks its children to reinsert the tree using *root* as their parent, by sending a $set-parent$ message. *root* already has 3 children, so it provides the cache back to the joining nodes, which contains

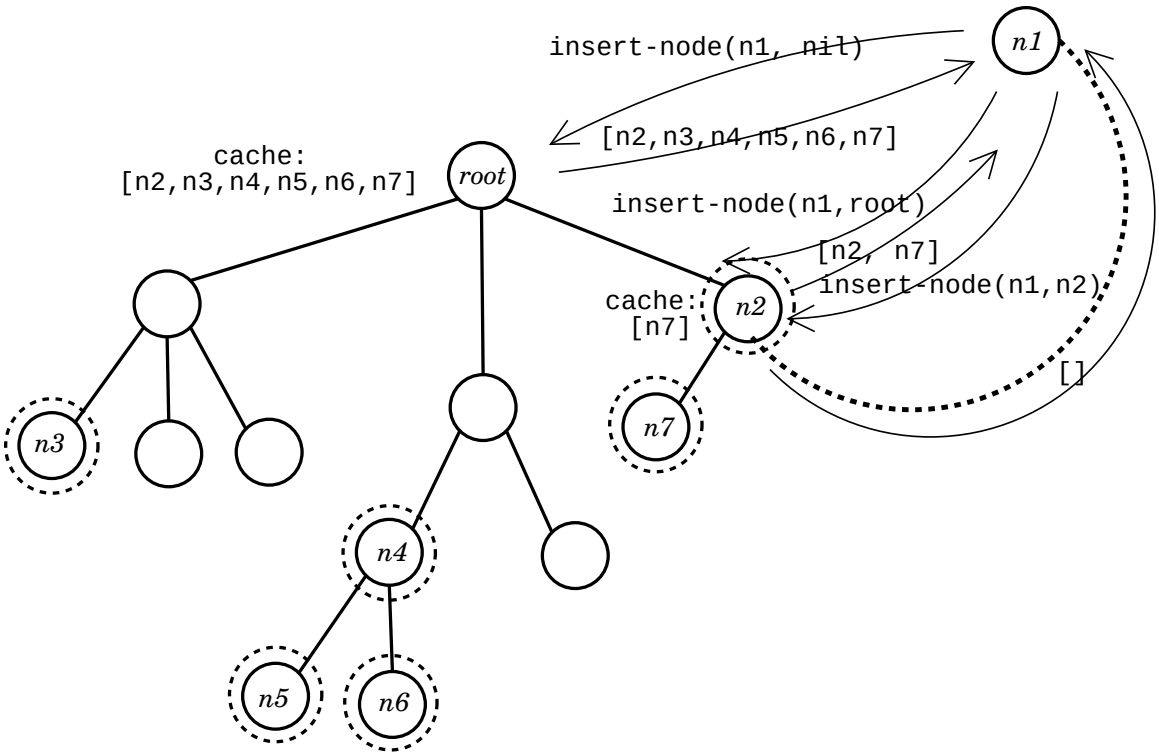

Figure 1—A node joining the multicast tree

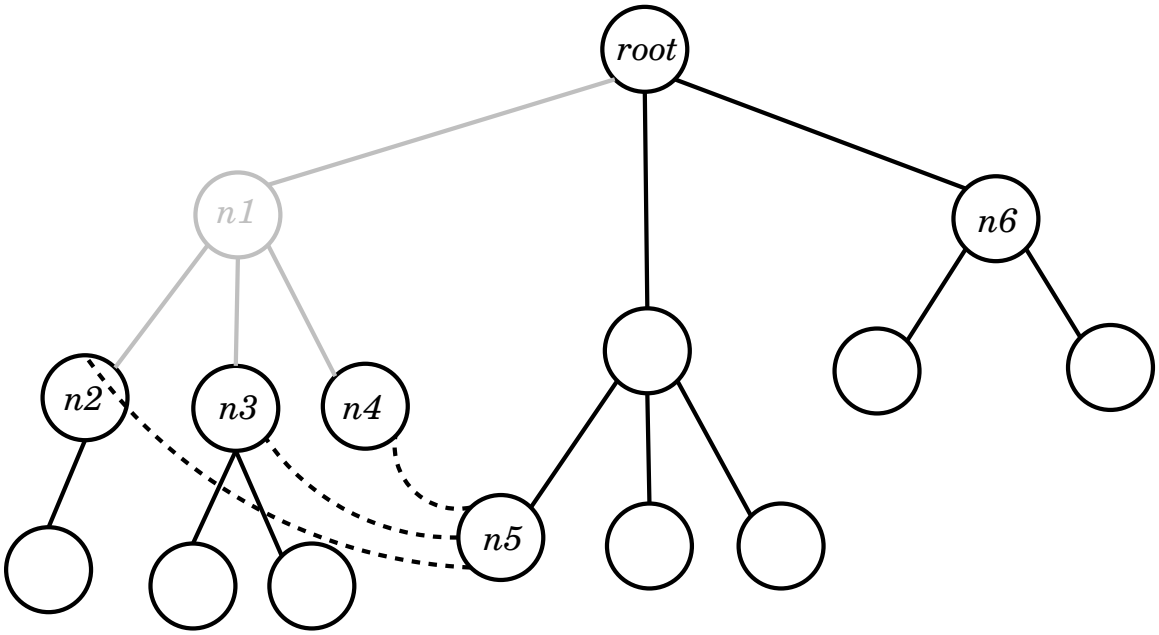

Figure 2—A node leaving the multicast tree and consequent reassignment

Algorithm 4 Leaving a group

```
node.leave( gid: groupid ):
    groups[gid].active = #f
    for n in groups[gid].children:
        n.set-parent( gid, groups[gid].parent
        groups[gid].children.remove( n )
    if groups[gid].parent not = nil:
        groups[gid].parent.erase-node( self )
    DHT.erase( gid, self )
    groups.remove( gid )
node.set-parent( gid: groupid, root: nodeid ):
    if root == nil:
        r = DHT.put!( gid, self, groups[gid].parent )
        if r.result:
            groups[gid].parent = nil
        else:
            self.set-parent( gid, r.value )
    else:
        if not self._insert( gid, root, nil ):
            self.set-parent( gid, nil )
node.erase-node( gid: groupid, nid: nodeid ):
    groups[gid].children.erase( nid )
    groups[gid].cache.erase( nid )
```

among others n_5. All 3 children determine that n_5 is their closest candidate parent, and according to the join algorithm they become children of n_5. Once the migration completes, n_1 asks *root* to forget it, and *root* drops n_1 as a child, completing the leave. It should be noted that while the migration is in progress, messages can continue to be routed in the tree, because the links are not broken until the operation has completed[1].

4 Experimental Evaluation

In this section, we experimentally evaluate the efficiency and cost of the groups constructed by the p2p objects group formation algorithms. The primary metric for the multicast tree setup is how well it is constructed in relation to the corresponding minimum spanning tree (MST). The MST is the optimal tree possible which minimizes the latency in sending messages over the tree. The weight of each edge between any two nodes is the latency between the nodes. Efficiency can then be estimated as the ratio between the sum of the weights of the edges in our tree compared to that of the optimal MST for that group.

A second metric is the relative delay penalty (RDP). RDP is defined as the ratio of the actual delay before a node receives the message and the unicast delay if the source sends the message directly to the recipient on the network. The RDP shows the relative speed of the multicast message dissemination.

In conventional IP multicast each node has a RDP of 1, since it is th most efficient distribution and follows the unicast path. Overlay networks generally introduce much higher RDPs since neighbors in the overlay network are not necessarily neighbors in the physical network.

We conduct our experiments using *p2psim*[2], a discrete event simulator for simulating P2P protocols. The simulator was set up with a random Euclidean topology. The latency is proportional to the distance between the nodes in the Cartesian co-ordinate system. There are no packet losses. The system is run with no queries and leaves to allow the algorithm to be easily evaluated.

[1] *_insert* in the join algorithm ignores the current parent and children of the node as candidate parents. This is not shown in Algorithm 3 for brevity.

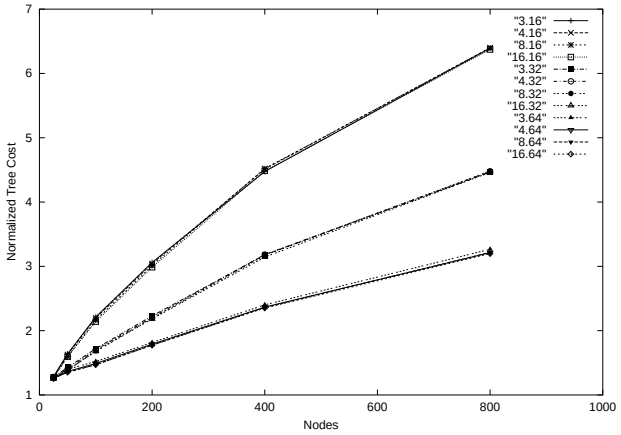

Figure 3—Effi ciency of the multicast tree compared to the MST

4.1 Communication Efficiency

We first evaluate the effect of different cluster sizes and cache sizes on the tree formed. Cluster size limits the number of children a node may have which might result in trees with long depth. We might expect this to increase the cost of the tree since there would be more number of hops on the average. But counter-intuitively, cluster size doesn't seem to make a difference. For the same cache size, varying cluster sizes doesn't make a difference in the cost of the tree constructed.

Each root stores a cache of nodes which recently joined the tree and the parents where they joined. When a new node joins, it looks at the cache and then selects the node closest to join. In the absence of a cache the node would have to recursively walk through the tree in order to find the best node. The cache helps in reducing this bootstrap time since the node has to look at a much smaller set of potential parents in order to make a good choice. The node can therefore quickly determine a new parent with significantly lesser number RPCs and join. Cache size thus makes a big difference in the cost of the tree. As we can see from Figure 3, doubling the cache size reduces the cost of the tree by a factor of 1.5 for a group size of 400. The improvement becomes better and better as the group size increases. This is expected since the algorithm does worse with increasing size. But still the cost grows sublinearly with system size. Hence even with groups of 800, the algorithm approximates the MST by a factor of 3.

The other communication primitive used is that of anycast. Anycast involves sending to the node closest to you. Efficient construction of multicast trees would result in efficient anycast, since nodes which are nearby in the latency space are likely to be connected to each other in the tree. The benchmark to compare again is the closest node in the MST. For each node in the group we compute the cost of sending an anycast message

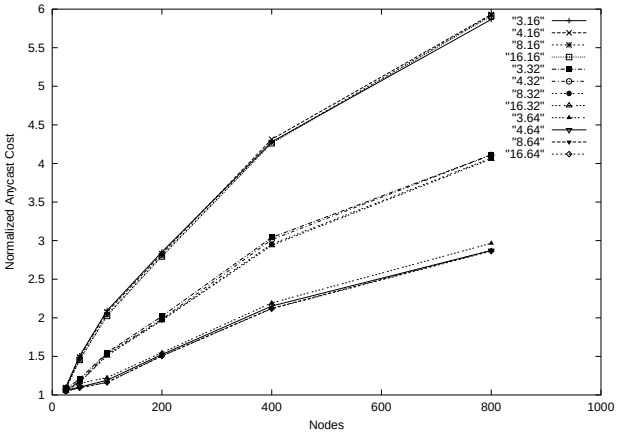

Figure 4—Effi ciency of doing anycast on our tree compared to MST

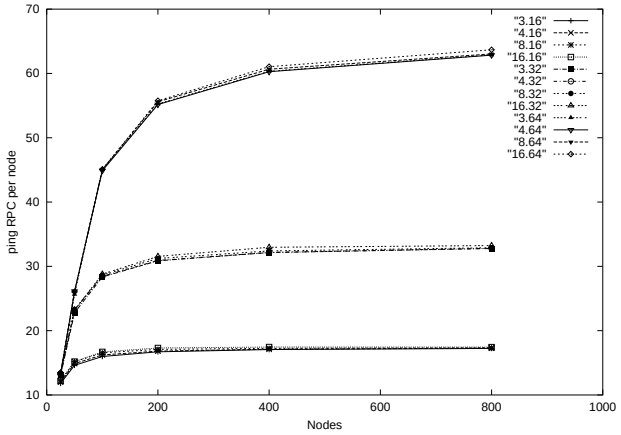

Figure 6—Overhead of determining the closest node

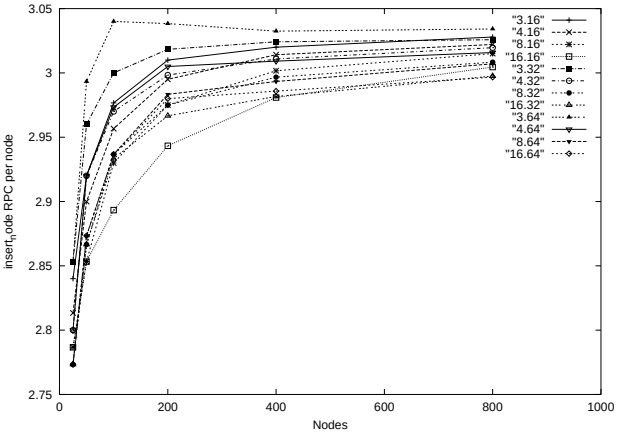

Figure 5—Cost of building the tree in terms of the number of RPCs made

and compare it with the corresponding cost in MST. We then average these ratios to calculate one metric for the algorithm. We plot this metric in Figure 4 for varying group sizes.

Cluster size again makes no difference to the cost, while cache size results in big differences. This is to be expected since as shown above, bigger cache size leads to more efficient multicast trees and correspondingly anycast will be efficient too.

4.2 Tree Formation Cost

When a node joins the group, it needs to find the appropriate place to join. It has to find a parent node as close to itself as possible. This is achieved by means of communicating with the root and obtaining a list of nodes who could be potential parents. It then tries them in order of increasing latency. Hence an important overhead is the number of RPC calls to different nodes in the process. We plot this cost in Figure 5. The number of RPCs are averaged per node. We plot the average with increasing group size. As we can see,the overhead converges to a constant value of around 3. Cluster size and cache size don't make a significant difference. This means that inspite of the increasing group size, the join overhead remains the same which is a very nice property.

An important step in picking the best node to join is finding the closest node in the node list the root returns to the new node. This involves pings to all the nodes in the list which the root returns. This is an important overhead involved in joining the tree and we quantify it in Figure 6. As expected, the number of pings approached the cache size as the system size increases.

Cache size thus seems to play two conflicting roles, it helps in building efficient trees, but a larger size results in a larger join overhead. In order to reduce the overhead, one possible heuristic which can be implemented is for the root to cache ping latencies in the cache and return it to the newly joining node. If there is a node in the cache which is in the same domain as the new node, the latencies are likely to be very similar and the new node can skip pinging these nodes. Alternatively, a network coordinate system like Vivaldi [7] can be used for maintaining accurate distance measurements between know nodes. We intend to explore these enhancements in the future.

4.3 Prototype Implementation

A prototype implementation of the P2P Objects group communications system is available, based on OpenHash[1]. The implementation consists of two SEDA [13] stages, one for supplying the DHT primitives required by the P2P Object algorithms, and another one for group management and message routing. The DHT stage uses request and response events for doing DHT lookups and performing the primitive operations. The group management stage communicates with application stages with 4 main event types, a join event, a leave event, a send message event, and a deliver message event for application upcalls. The implementation has been tested in small scale settings; we plan to build a full-fledged application to be deployed and tested in the PlanetLab[3] testbed.

5 RELATED WORK

Search in P2P systems has been an area of active research. The PIER [11] project at Berkeley is building a massively distributed query engine based on DHTs. The project focuses on being a generic database engine based on DHTs. The use the starting few bits of the DHT namespace to identify the type of the resource. A search of a particular item belonging to a particular resource type is essentially multicast on the key-range on which that resource type could be present. But they assume the

underlying DHT layer to provide the routing and focus mainly on the database aspects. Hence their multicast primitive is not optimized. Also a resource type which has a huge number of objects belonging to it, will result in a lot of overhead in querying. The definition of resource type is also static.

Multicast was originally introduced as an IP based solution. Due to the difficulty of deployment, several recent proposals have argued for application level multicast. Narada [10] provides small scale multicast groups. End systems organize themselves into a mesh structure using a distributed protocol. But the system does not scale beyond hundreds of nodes and does not perform well under dynamic conditions due to the use of tree forming algorithms.

Recent proposals have used P2P routing algorithms for efficient multicast in the presence of dynamic groups. Most of the proposals use a DHT to hash the group name to a particular key which maps to a particular node. The node then forms the Rendezvous point (RP) and reverse path routing is then used to form the multicast tree. Several such proposals based on the different flavors of DHTs in vogue have been mooted. For example [5, 6] use Pastry [8] as the underlying DHT to implement multicast and anycast, by routing through the DHT.

Coral [9] describes the concept of a sloppy hash-table. This is a modification of DHT, to provide a $1 \rightarrow M$ instead of a $1 \rightarrow 1$ mapping. This can provide the mechanism for multicast and anycast communication, although the burden is still in the application writer.

Nonetheless the above multicast primitives are different from the one necessary for implementing efficient querying and resource discovery. These proposals are either in the traditional multicast mold, essentially they are enrollment based, or they perform indirect routing through the DHT. Both these architectures are unsuitable for querying large scale distributed systems since there is no relation between the group formed and the nature of objects/data the nodes actually contain.

6 Conclusion and Future Work

In this paper, we presented P2P Objects, a system designed for large scale querying in distributed systems based on group communications primitives. The main contribution of this work is a novel multicast tree construction and maintenance algorithm and an application architecture for intelligent queries based on multicast and *k*-cast group communication primitives. We presented our algorithms in detail, evaluated their performance using simulations, and provided a real world implementation.

As part of future work, there are some interesting directions we would like to explore. We are considering using a sophisticated coordinate system [7] for distance metrics, and enhancements in the routing algorithm that allows response aggregation for reply messages. An important aspect of the system that still remains to be evaluated is how it responds to failures and what the correct mechanisms for repairing trees are. We are actively working in expanding the system to include a stabilization algorithm which improves the quality of the tree over time and also allows us to deal with failures in a consistent fashion. Finally, we are planning to test the system's implementation in the PlanetLab testbed, and complete a sample application that uses our communication primitives.

References

[1] Openhash - a robust open source dht. http://openhash.org.

[2] p2psim - a discrete event simulator for p2p networks. http://pdos.lcs.mit.edu/p2psim.

[3] Planetlab - internet scale testbed. http://planet-lab.org.

[4] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth Symposium on Architectural Supports for Programming Languages and Operating Systems (ASPLOS V)*, 1992.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. In *IEEE Journal on Selected Areas in Communication (JSAC)*, October 2002.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networks and Group Communications (NGC)*, September 2003.

[7] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[8] P. Druschel and A. Rowstron. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[9] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[10] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM Sigcomm*, 2001.

[11] R. Huebsch, J. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

[12] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Transactions on Networking*.

[13] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

[14] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-ar ea location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U.C. Berkeley, 2001.