Optimizing Wide-Area Application Level Multicasting using P2P Resource Pool

Zheng Zhang
Xing Xie
Shi-Ding Lin
Bo-Ying Lu

May 16, 2003

Technical Report
MSR-TR-2003-36

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Optimizing Wide-Area Application Level Multicasting using P2P Resource Pool

Zheng Zhang, Xing Xie, Shi-Ding Lin, Bo-Ying Lu
Microsoft Research Asia
5F, Sigma building, No.49, Zhichun Road
Beijing, 100080, P.R.China
{zzhang, xingx, i-slin, t-bylu}@microsoft.com

ABSTRACT

Wide-area application level multicasting (ALM) has been an active research area lately and practical algorithms have been derived to deliver QoS results for small to medium group size, covering a good range of cases when such requirements are warranted. These algorithms all assume that the resulting tree is comprised of the members in the session *only*. However, in a large and collaborative environment, active sessions are likely to consume only a fraction of the total resources. Moreover, some nodes with large capacity can contribute to multiple sessions. While it is intuitive to explore such spare resources, the challenges are 1) how to organize all the available resources into a resource pool and 2) how to discover and subsequently utilize the spare resources to benefit the active sessions.

In this paper, we describe SOMO (*Self-Organizing Metadata Overlay*) which is an in-system monitoring service that effectively creates an illusion of a single resource pool made up by machines organized using P2P technologies. Using SOMO, we show practical solutions utilizing spare resources can substantially optimize active ALM sessions. Furthermore, sessions with different priorities occupy resources accordingly. All these are achieved using a hybrid model that combines in-time global knowledge and individual competition without the need of central coordination.

1. INTRODUCTION

Application-level multicasting (ALM) is one of the most interesting applications of *overlay network*. It happens to present many challenges as well: for scenarios such as video-conferencing, guaranteeing certain QoS metrics is of paramount importance.

Many algorithms have been proposed to address these problems, all assuming that the only resources available are those in the ALM session. In a collaborative environment, many other stand-by resources could be otherwise included for a more optimal solution. For example, Microsoft Research has five branches across the globe, and has many thousands of machines that are geographically distributed. At a given hour, however, number of active sessions is likely to be only a handful, and each session may have a small number of participants (say less than 20). Thus, a strong case can be made to orchestrate all the resources together so that active sessions can utilize spare resources when beneficial.

While this idea is rather intuitive, the challenges are many. There are two critical building blocks: 1) how to organize a *resource pool* and 2) how to schedule sessions by recruiting spare resources, and do so in a completely *distributed* manner.

To this end, we make a few novel contributions in this paper:

- We use the latest P2P technologies to self-organize potentially very large amount of resources. In particular, we employ P2P DHT (Distributed Hash Table) to pool resources together. However, pooling resources does not automatically yield a resource pool, yet.
- Extending our early work of [20], we demonstrate the feasibility of an infrastructure embedded in arbitrary P2P DHT that provides a highly efficient, robust and scalable monitoring service. This infrastructure SOMO (*Self-Organized Metadata Overlay*) is fault-resilient and can gather and disseminate system information in O(log*N*) time. In essence, SOMO builds a dynamic system status database which is available internally to system participants. This database is being continuously updated and creates an illusion of a single, large resource pool.
- We then demonstrate, step by step, how ALM sessions can be optimized by finding spare resources in the pool. We first show how this can be done assuming only a single session is of interest, and validate that up to 30% improvement can be made for small-to-medium group size. All data necessary for making scheduling decision are gathered through SOMO and then subsequently queried at the time of scheduling. We then extend the base algorithm to schedule multiple, *simultaneous* sessions each may of different priorities. To ensure scalability, we take cues from sociology and adopt a simple model in which individual, credential-based competition is combined with on-time global knowledge available through SOMO. Our results show that, as

expected, sessions of higher priority are given higher share of resources, resulting better performance

While our work is targeted at optimizing ALM sessions, the core technology is a lot more generic: we show how a resource pool can be efficiently built, with no administration overhead; and the philosophy and model of distributed job scheduling – borrowing ideas from time-tested practice in society, can be applied to other distributed applications.

The rest of the paper is organized as follows. How to construct the resource is the focus of Section 2. Scheduling multiple ALM sessions in the resource pool created by SOMO is extensively studied in Section-4. We discuss related work in Section-4 and conclude in Section-5.

2. BUILDING P2P RESOURCE POOL

The foundation of our resource pool proposal is the so-called *structured* peer-to-peer systems, and in particular the *distributed hash table* (DHT). We assume that the readers are reasonably familiar with the concept of DHT, and for the sake of brevity will only go through the basics.

In DHT, a very large logical space (e.g. 160-bits) is assumed. Nodes join this space with random IDs and thus partition the spaced uniformly. The ID can be, for instance, MD5 over a node's IP address. An ordered set of nodes, in turn, allows a node's responsible *zone* to be strictly defined. Let p and q be a node x's predecessor and successor, respectively. One definition of a node's zone is simply the space between the ID of its immediate predecessor ID (non-inclusive) and its own ID. In other words: $zone(x) \equiv (ID(p), ID(x)]$. This is essentially how consistent hashing assigns zones to DHT nodes [16] (Figure 1). This base ring (also called as *leaf-set*, as in Pastry[13]) is the simplest P2P DHT. To harden the ring against system dynamism, each node records r neighbors to each side. These states are the basic routing table, and are updated to keep the invariant when node join/leave events occur.

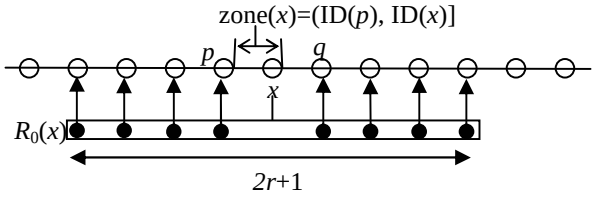


Figure 1: the simplest P2P DHT – a ring, the *zone* and the basic routing table that records *r* neighbors to each side.

If one imagines the zone being a hash bucket in an ordinary hash table, then the ring is a *distributed hash table*. Given a key in the space, one can always resolve which node being responsible. The lookup performance is O(N) in this simple ring structure.

Elaborate algorithms built upon the above concept so that they achieve O(log$N$) performance with either O(log$N$) or even constant states (i.e. the routing table entries). Representative systems including Chord[16], CAN[10], Pastry[13] and Tapestry[22].

The most interesting aspect of a DHT is that the whole system is self-organizing with very low overhead – typically in the order of O(log$N$). The second significant attribute is the virtualization of a space where both resources and other entities (such as documents stored in DHT) live together; this feature is what we explore the most in this paper.

Many DHT systems are designed with a storage-centric mindset. We found it more interesting simply to take advantage of DHT's capability of stringing together large amount of resources *without* administration oversight. However, pooling resources together does not automatically yield a resource pool. A resource pool exists so that resource sharing at the time of scheduling tasks (e.g. application-level multicast sessions) is possible. This requires two more pieces:

1. An efficient way to know the running states of the resources in the pool. And,
2. Based on 1), a methodology to schedule an incoming task.

Therefore, embedded inside the system itself, there must be a robust, highly efficient and scalable monitoring utility that can gather and disseminate global knowledge as accurately as possible. This is so because for a large system, it is impractical to rely on external monitoring service.

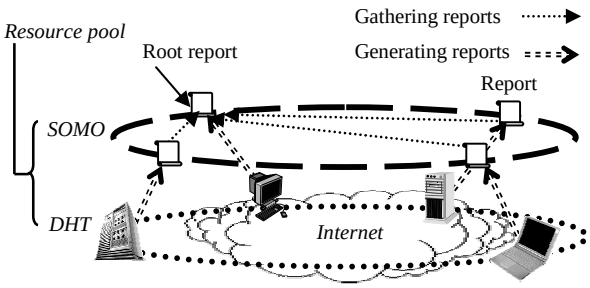


Figure 2: the resource pool is comprised of the machines pooled together via DHT. SOMO, a self-organizing hierarchy using data overlay that efficiently aggregates resource status in a scalable way, is an in-system monitoring utility. Combining DHT's capability of pooling resource with SOMO collectively makes the resource pool.

This in-system monitoring utility is called *self-organizing metadata overlay* (SOMO), and will be introduced in Section 2.2. SOMO is built using a generic technology, *data overlay*, which can construct *arbitrary* distributed data structure over a DHT. The relationship of these concepts is described in Figure 2.

2.1 Data Overlay

We observe that hash-table is only one of the fundamental data structures. Sorted list, binary trees and queues etc. all have their significant utilities. One way would be to investigate how to make each of them self-organizing (i.e., P2P sorted list). Another is to build on top of a hash table that already has such self-organizing property (i.e. P2P DHT). This second approach, which we call *data overlay*, is what we take in this paper.

Any object of a data structure can be considered as a document. Therefore, as long as it has a key, that object can be deposited into and retrieved from a P2P DHT. Objects relate to each other via pointers, so to traverse to object *b* pointed to by *a.foo*, *a.foo* must now store *b*'s key instead. More formally, the following two are the necessary and sufficient conditions:

- Each object must have a key, obtained at its birth
- If an attribute of an object, *a.foo*, is a pointer, it is expanded into a structure of two fields: *a.foo.key* and *a.foo.host*. The first substitutes the hard-wired address of pointer, and the second field is a soft state containing the last known hosting DHT node of the object *a.foo* points to and serves as a routing shortcut.

It is possible to control the generation of object's key to explore data locality in a DHT. For instance, if the keys of *a* and *b* are close enough, it's likely that they will be hosted on one machine in DHT.

We call a data structure distributed in a hosting DHT a *data overlay*. It differs from traditional sense of overlay in that traversing (or routing) from one entity to another uses the free service of the underlying P2P DHT.

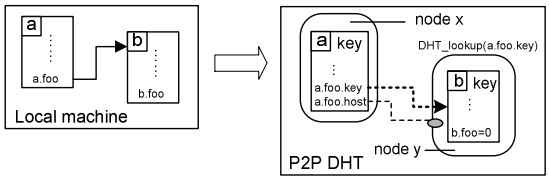


Figure 3: implement arbitrary data structure in DHT.

Figure 3 contrasts a data structure in local machine versus that on a P2P DHT. Important primitives that manipulate a pointer in a data structure, including *setref*, *deref* (dereferencing) and *delete*, are outlined in Figure 4. Here, we assume that both DHT_lookup and DHT_insert will, as a side effect, always return the node in DHT that currently hosts the target object. DHT_direct bypasses normal DHT_lookup routing and directly seeks to the node that hosting an object given its key.

The interesting aspect is that it is now possible to host any arbitrary data structure on a P2P DHT, and in a transparent way. The *host* routing shortcut makes the performance insensitive to the underlying DHT.

```
setref(a.foo, b) {     // initially a.foo==null; b is the object
                       // to which a.foo will points to
  a.foo.key=b.key
  a.foo.host= DHT_insert(b.key, b)
}
deref(a.foo) {         // return the object pointed to by a.foo
  if (a.foo≠null) {
    obj=DHT_direct(a.foo.host, a.foo.key)
    if obj==null {     // object has moved
      obj=DHT_lookup(a.foo.key)
      a.foo.host = node returned
    }
    return obj
    else return "non-existed"
  }
}
delete(a.foo) {        // delete the object pointed to by a.foo
  DHT_delete(a.foo.key)
  a.foo=null
}
```

Figure 4: pointer manipulate primitives in data-overlay

A data overlay on top of a bare-bone P2P DHT with no internal reliability support can be used to implement distributed data structure that is soft-state in nature (i.e,, data is periodically refreshed and consumed thereafter without ill side-effect). This is adequate to monitor the running state of resource pool as a whole, and is what we employ for SOMO.

2.2 SOMO: Self-Organized Metadata Overlay

We now describe the data overlay SOMO (*Self-Organized Metadata Overlay*), a generic information gathering and disseminating infrastructure on top of any P2P DHT. In a way, SOMO can be thought as a responsive "news broadcast" whose construction and processing are shared by all the nodes. The on-time "news" is what creates the illusion of the resource pool.

Such an infrastructure must satisfy a few key properties: *self-organizing* at the same scale as the hosting DHT, fully *distributed* and *self-healing*, and be as *accurate* as possible of the metadata gathered and disseminated.

Such metadata overlay can take a number of topologies. For the sake of resource pool, one of the most important functionalities is aggregation. Therefore, our implemented SOMO is a tree of *k* degree whose leaves are planted in each DHT node. Information is gathered from the bottom and propagates towards the root, and disseminated by trickling downwards. Thus, one can think of SOMO as doing *converge cast* from the leaves to the root, and then (optionally) *broadcast* back down to the leaves again. Both the gathering and dissemination phases are $O(\log_k N)$

bounded, where N is total number of entities. Each operation in SOMO involves no more than k+1 interactions, making it fully distributed. We deal with robustness using the principle of soft-state, so that data can be regenerated in $O(\log_k N)$ time. The SOMO tree self-organizes and self-heals in the same time bound.

Since SOMO is a tree, we call its node the *SOMO node*. To avoid confusion, we denote the DHT nodes as simply the *DHT node*. At this point, it is worth to emphasize that SOMO is a *data overlay* and as such is a distributed data structure spread onto the DHT. A SOMO node is an object as in object-oriented programming language, and its member functions will be carried out by its hosting DHT node (i.e., the machine).

2.2.1 Building SOMO

A DHT node that hosts a SOMO node *s*, is referred to as *DHT_host*(*s*).

```
struct SOMO_node {
  string key
  struct SOMO_node *child[1..k]
  DHT_zone_type Z
  SOMO_op op
  Report_type report
}
```

Figure 5: SOMO node data structure

The basic structure of the type *SOMO_node* is described in Figure 5. The member *Z* indicates the region that this node's *report* member covers. Here, the region is simply a portion of the total logical space of the DHT. The root SOMO node covers the entire logical space. The *key* is produced by a deterministic function of a SOMO node's region *Z*. Examples of such functions include the center of the region, or a hash of the region coordinates (see Figure 6). Therefore, a SOMO node *s* will be hosted by a DHT node that covers *s.key* (e.g. the center of *s.Z*). This allows a SOMO node to be retrieved deterministically – exactly the same as any other documents stored in DHT, as long as we know its responsible region, and is particularly useful when we want to query system status in a given key-space range. A SOMO node's responsible region is further divided by a factor of *k*, each taken by one of its *k* children, which are pointers in the SOMO data structure. A SOMO node *s*'s *i*-th child will cover the *i*-th fraction of region *s.Z*. This continues recursively until a termination condition is met (discussed shortly). Since a DHT node will own a piece of the logical space, it is therefore guaranteed a SOMO node will be planted in it.

Initially, when the system contains only one DHT node, there is only the SOMO root. As the DHT system grows, SOMO builds its hierarchy along. This is done by letting each DHT node periodically execute the routine *SOMO_grow* shown in Figure 6, for any SOMO nodes that are in its custody.

```
SOMO_grow(SOMO_node s) {
        // check if any children is necessary
 if (s.Z⊆DHT_host(s).Z) return
 for i=1 to k
  if (s.child[i]==NULL &&
      the i-th sub-space of s.Z ⊄ host(s).Z) {
     t = new(type SOMO_node)
     t.Z = the i-th sub-space of s.Z
     t.key = SOMO_loc(t.Z)
     setref(s.child[i], t)        // inject into DHT
  }
}
SOMO_loc(DHT_zone_type Z) {
 return center of Z
 // optionally
 // return hash_of (Z)
}
```

Figure 6: SOMO_grow procedure and the SOMO_loc procedure which deterministically calculates a SOMO node's key given the region it covers. The procedure is executed by the hosting DHT machine.

We test first if the SOMO node's responsible zone is smaller or equal to that of the hosting DHT node, if the test comes out to be true, then this SOMO node is already a leaf planted in the right DHT node and there is no point to grow any more children. Otherwise, we attempt to grow. Note that after a new SOMO node is initialized, we call the *setref* primitive (See Figure 4) to install the pointer; this last step is where DHT operation is involved. This way, new SOMO nodes covering smaller regions are installed into the DHT.

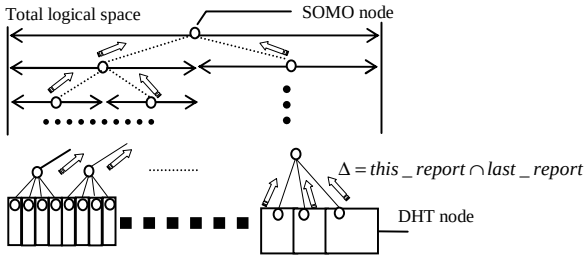


Figure 7: SOMO tree on top of P2P DHT. Circles are SOMO nodes. SOMO nodes are mapped onto DHT nodes according to their keys. A DHT node may own multiple SOMO nodes and will execute their *SOMO_grow* routines periodically.

As this procedure is executed over all SOMO nodes, the SOMO tree will grow as the hosting DHT grows, and the SOMO tree is taller in logical space regions where DHT nodes are denser. This is illustrated in Figure 7. Note that SOMO nodes fall on to DHT nodes according to their keys. As such a DHT node may own more than one SOMO node, but has at least one SOMO node planted into it.

The *SOMO_grow* procedure is done in a top down fashion, and is executed periodically. A bottom-up version can be similarly derived. When the system shrinks, SOMO tree will prune itself accordingly by deleting redundant

children. For an *N*-node system where nodes populate the total logical space evenly, there will be 2*N* SOMO-nodes when the SOMO fan-out *k* is 2.

The crash of a DHT node will take away the SOMO nodes it is hosting. However, the crashing node's zone will be taken over by another DHT node after repair. Consequently, the periodical checking of all children SOMO nodes ensures that the tree can be completely reconstructed in $O(\log_k N)$ time. Because the SOMO root is always hosted by the DHT node that owns one deterministic point of the total space, that node ensures the existence of the SOMO root and invokes the SOMO_grow routine on the SOMO root.

2.2.2 Gathering and Disseminate Information with SOMO

To gather system metadata, for instance loads and capacities, a SOMO node periodically requests report – executed by DHT node that holds it from its children. The leaf SOMO nodes simply get the required info from their hosting DHT nodes. As a side-effect, the procedure will also re-start a child SOMO node if it has disappeared because the hosting DHT node's crash. Figure 8 illustrates the procedure.

```
get_report (SOMO_node s) {
  Report_type rep[1..k]
  for i∈[1..k]
    if (s.child[i] ≠ NULL)        // retrieving via DHT
      rep[i] = deref(s.child[i]).report
  s.report = s.op(rep[])
}
```

Figure 8: SOMO gathering procedure, executed by DHT nodes responsible for a SOMO node.

The routine is periodically executed at an interval of *T*. Thus, information is gathered from the SOMO leaves and flows to its root with a maximum delay of $\log_k N \cdot T$. This bound is derived when flow between hierarchies of SOMO is completely unsynchronized. If upper SOMO nodes' call for reports immediately triggers the similar actions of their children, then the latency can be reduced to $T+t_{hop}\cdot\log_k N$, where t_{hop} is average latency of a trip in the hosting DHT. The unsynchronized flow has latency bound of $\log_k N \cdot T$, whereas the synchronized version will be bounded by *T* in practice (e.g., 5 minutes). Note that $O(t_{hop} \cdot \log_k N)$ is the absolute lower bound. For 2M nodes and with k=8 and a typical latency of 200ms per DHT hop, the SOMO root will have a global view with a lag of 1.6s.

If the SOMO report is composed of information pertain to building the resource pool, such as load and network condition of the machine, then by continuing to gather fresh report from the SOMO leaves (and thus *every* machines in the pool), SOMO root will have periodical snapshots of the whole system. Such snapshot's may contain information whose freshness is $O(\log N)$ bounded, and is adequate for scheduling coarse-grained jobs such as application-level multicasting which usually last much longer than the collection period.

Dissemination using SOMO is essentially the reverse: data trickles down through the SOMO hierarchy towards the leaves. Performance thus is similar as gathering. The other alternative is to query the SOMO root. This is what we used in scheduling ALM sessions, since number of sessions is relatively small and the query is not made very often.

Operations in either gathering or disseminating phases involve one interaction with the parent, and then with *k* children. Thus, the overhead in a SOMO operation is a constant. The entities involved are the DHT nodes that host the SOMO tree. SOMO nodes are scattered among DHT nodes and therefore SOMO processing is distributed and scales with the system.

It seems that towards the SOMO root the hosting DHT nodes need to have increasingly higher bandwidth and stability. As discussed earlier, stability is not a concern because the whole SOMO hierarchy can be recovered in $O(\log_k N)$ time. As for bandwidth, most of the time one needs only to submit delta between reports (Figure 8). Combining with compression will further bring down message size. Finally, it is always possible to locate an appropriate DHT node through SOMO. This node can swap with the one who is hosting the SOMO root currently. That is to say, SOMO can be completely self-optimizing as well.

The power of SOMO lies in its simplicity and flexibility: it specifies neither the type of information it should gather and/or disseminate, nor the operation invoked to process them. That is to say, SOMO operations are programmable and *active*. For this reason, in the pseudo-code we have used *op* as a generic notation for operation used. Using the abstraction of data overlay, its performance is also insensitive to the hosting DHT. SOMO processing is fully distributed, and it is both self-organizing and self-healing.

We have implemented a SOMO-based global performance monitor with which we monitor the servers in our lab on a daily basis. This tool employs SOMO built over a very simple ring-like DHT, and SOMO gathers data from various performance counter on each machine. The complete system status is obtained by querying the SOMO root report through a unified UI interface. We tested the SOMO stability by unplugging cables of servers being monitored, and each time the global view is regenerated after a short jitter. Using the data overlay abstraction, the SOMO layer is implemented much like any local procedures, with only a few hundred lines of code.

2.3 DHT+SOMO Ł P2P Resource Pool

To summarize, our P2P resource pool is composed of two ingredients:

- **DHT**. A DHT is used not in the sense of sharing contents, but rather as an efficient way to pool together a very large amount of resources, with zero administration overhead and no scalability bottleneck.
- **SOMO**. Utilizing the fact that arbitrary distributed data structure can be built in the virtual space and then mapped on to various resources, SOMO is a self-organizing "news broadcast" hierarchy. Aggregating resource status in O(log*N*) time then creates the illusion of a single resource pool.

3. SCHEDULING ALM SESSIONS WITHIN THE P2P RESOURCE POOL

As discussed in Section-2.2, built upon DHT, SOMO can create the image of a single resource pool. Given that, the interesting question is how job scheduling can be performed over this resource pool, in a completely distributed fashion with the goal of maximizing resource utilization *and* satisfying QoS requirements per application.

For the particular problem of application level multicasting (ALM), the end goal is for active sessions to achieve optimal performance with available resources in the pool. Session's performance metrics is determined by certain QoS definitions. Moreover, higher priority sessions should proportionally acquire more shares of resources.

We will give our QoS definition and then describe our approaches in steps. First, we will show how additional resources are recruited assuming only one single session is active. Next, we will present our approach of how multiple sessions with different priorities are optimized.

Unless otherwise specified, our experiments simulate a two-layer Transit Stub topology [18] with 600 routers. The network consists of 24 transit routers and 576 stub routers. We assign link latencies of 100ms for intra-transit domain links, 25ms for stub-transit links and 10ms for intra-stub domain links. We also append 1200 end systems to the stub routers randomly and set the last hop latency to a random value between 3ms and 8ms. The resource pool contains all the 1200 end nodes in the network. Similar to many previous works [14][15][19], each node has a bound on the number of communication sessions it can handle, which we call *degree*. This may due to the limited access bandwidth or workload of end systems. The degree bound for all the nodes lie within 2 and 9, and follows the distribution 2^{-i} for degree i-1. Thus, half of the nodes in the system have degree 2 and the population for higher degree decreases exponentially. These nodes are organized using a DHT and runs SOMO on top. The details of SOMO report for scheduling ALM will be presented later.

3.1 ALM QoS Definition

Each multicasting session assumes static membership, i.e., $M(s)$ is known a priori. We believe this covers a good portion of ALM applications where QoS is desired, for instance pre-scheduled video-conferencing.

For ALM, there exist several different criteria for optimization, like bandwidth bottleneck, maximal latency or variance of latencies. In this paper, we choose maximal latency of all members as the main objective of tree building algorithms since it can greatly affect the perception of end users. Our definition of QoS for one given session is the same as proposed in AMCast [15] and can be formally stated as follows:

Definition 1. Degree-bounded, minimal height tree problem (DB-MHT). Given an undirected complete graph $G(V,E)$, a degree bound $d_{bound}(v)$ for each $v \in V$, a latency function $l(e)$ for each edge $e \in E$. Find a spanning tree T of G such that for each $v \in T$, degree of v satisfies $d(v) \leq d_{bound}(v)$ and the height of T (measured as aggregated latency from the root) is minimized.

Using the resource pool, the above definition is slightly extended. An extended set of helper nodes H is added to the graph, and our objective is to achieve the best solution relative to an optimal plan derived *without* using H, by adding the least amount of helper nodes.

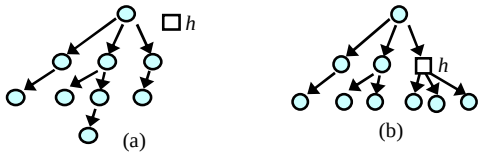


Figure 9: (a) an optimal plan for an ALM. (b) an even better plan using helper nodes in the resource pool. Circles are members belong to *M*(*s*), and the square is an available node with a large degree.

Figure 9 depicts this graphically. Suppose P_0 is the optimal plan by some algorithm, *f*, which involves the initial member set $M(s)$ only. Running a modified algorithm *f'* which not only uses $M(s)$ but also recruits available and nearby large degree nodes, the tree height – which corresponds to the maximal latency of the session, can be substantially reduced.

When the group size is very small (e.g., $|M(s)| \leq 10$), finding P_0 by enumerating all possibilities can be done in reasonable time: for an eight-node group this is less than three seconds on a 1.4GMHz Pentium IV PC with 256M memory. Even with this globally optimal plan, it is interesting to see that one helper node will still be able to shorten the tree. For instance, adding a helper node close

to the root with a degree bound greater than 4 brings about 14.2% average latency reduction. This validates our basic premise; the challenge is how to extend it for larger groups.

3.2 Scheduling a Single Session

We assume the root of the tree is where the planning and scheduling is performed. In other word, the root is the *task manager* of the session.

```
ALM(r, V) {            // V==M(s), r is the root
  for all v∈V          // initialization
    height(v)=l(r, v); parent(v)=r
  T = (W={r}, Link={})

  while (W<V) {        // loop until finish
    find u∈{V-W} s.t. height(u) is minimum
    if (d(parent(u))==d_bound(parent(u))-1)
      h=find_helper(u)
    if h≠NULL {        // integrate the helper node
      W+={h}; Link+={h, parent(u)};
      W+={u}; Link+={u, h};
    } else
      W+={u}; Link+={u, parent(u)};

    for all v∈{V-W} {  // re-adjust the height
      height(r)=∞
      for all w∈W
        if d(w)<d_bound(w) && height(v)>height(w)+l(w, v)
          height(v)=height(w)+l(w, v); parent(v)=w
    }
  }
  adjust(T)
  return T
}
```

Figure 10: the AMCast algorithm that is O(N^3) (without the lines in the dashed box) and the critical-node algorithm that utilizes additional helper node.

There are many proposals to optimize DB-MHT, which is generally known as NP-complete. Our goal is not to propose new algorithms. Instead, we will select from a few well-known ones and investigate how much performance benefits we can achieve when the resource pool is utilized.

Our base algorithm uses the one proposed in [15], with O(N^3) performance bound. This algorithm can generate a solution for hundred of nodes in less than one second (Figure 10, without the code in the dashed box). This algorithm, which we refer to as "AMCast," is a typical greedy algorithm. It starts first with the root and adds it to a set of the current solution. Next, the minimum heights of the rest of the nodes are calculated by finding their closest potential parents in the solution set, subject to degree constraints. This loops back by absorbing the node with the lowest height into the solution. The process continues until all nodes are finally included in the resulting tree.To ensure that we get the best possible tree to start with, we augment this algorithm with further tuning (line 21).

A known technique to approximate globally optimal algorithm is to adjust the tree with a set of heuristic moves. These moves are graphically depicted in Figure 11, The adjustments include the followings: (a) find a new parent for the highest node; (b) swap the highest node with another leaf node; (c) swap the sub-tree whose root is the parent of the highest node with another sub-tree. These optimizations are local adjustment after the tree is generated using the AMCast algorithm, and will be referred to as *adjust*. In our experiments, we test how helper nodes affect the algorithm both with and without this improvement.

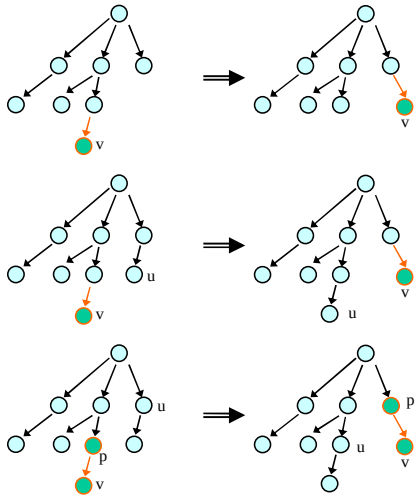


Figure 11: Adjustment heuristics for DB-MHT. From top to bottom: find a new parent for the highest node; swap the highest node with another leaf node; swap the sub-tree whose root is the parent of the highest node with another sub-tree.

Our algorithm searching for beneficial helper nodes include two considerations: the time to trigger the search and the criteria to judge an addition. The general mechanism is described by the pseudo-code in the dash-box of Figure 10. Let u be the node that the AMCast algorithm is about to add and *parent*(u) be its parent. When *parent*(u)'s free degree is reduced to one, we trigger the search for an additional node h. If such h exists in the resource pool, then h becomes u's parent instead and replaces u to be the child of the original *parent*(u). Different versions vary only on the selection criteria of h but we refer to this class of optimization the *critical node* algorithm. "Critical" here means that, for a particular node, this is the last opportunity to improve upon the original greedy algorithm.

We have experimented with different algorithm to search for h. The first variation is simply to find an additional node closest to the parent node and with an adequate

degree (we use 4). Let $l(a, b)$ be latency between two arbitrary nodes a and b. We find the following heuristic yields even better results:

$l(h, parent(u))+\max(l(h, v))$ is minimum

where v satisfies $parent(v) = parent(u)$ && \\ condition 1

$d_{bound}(h) \geq 4$ && \\ condition 2

$l(h, parent(u))<R$ \\ condition 3

Here, v maybe one of u's siblings. The idea here is that since all such v will potentially be h's future children, $l(h, parent(u)) + \max(l(h, v))$ is most likely to affect the potential tree height after h's joining (condition 1). Such helper node should have adequate degree (condition 2). Finally, to avoid "junk" nodes that are far away even though their degrees are high, we impose a radius R: h must lie within R away from $parent(u)$ (condition 3).

We found that R between 50~150 yields satisfactory results for the topology parameters we chose. The tradeoff here is that a small R will reduce the choice of candidates, whereas a larger R will introduce links of long latency in the tree. That the setting of radius to be medium range gives good result isn't a surprise. Recall that we have 100, 25, and 10 for intra-transit, stub-transit and intra-stub links respectively. Thus, a radius of 50-150 will avoid all nodes from another stub.

So far we have described the algorithm as if we not only knew the degrees of other nodes in the resource pool, but also the latencies between all pairs. While the first condition can be easily met by querying the SOMO reports, the 2nd is obviously impractical because it entails, virtually, that latencies between all pairs are available. To get around this problem, we use the well known "landmark" approach. In this algorithm, a few landmarks are chosen first. Each node then measure roundtrip times to these landmarks and the resulting *delay vector* approximates the coordinates of a node relative to the landmarks. To judge the closeness of two nodes, the Euclidean distances between the two delay vectors are computed.

In our experiments, we chose three random landmarks. Each node now includes its measured delay vector and its degree when submitting to SOMO. To find the closest helper node h to $parent(u)$ at the time of searching, we use delay vector of $parent(u)$ to query the SOMO report for 5 nearby nodes and then select the one that fits the criteria the best.

In the followings, we call the algorithm where pair-wise node latency is known a priori via an oracle the *Critical*, and the one used the landmark estimation for vicinity judgment the *Landmark*.

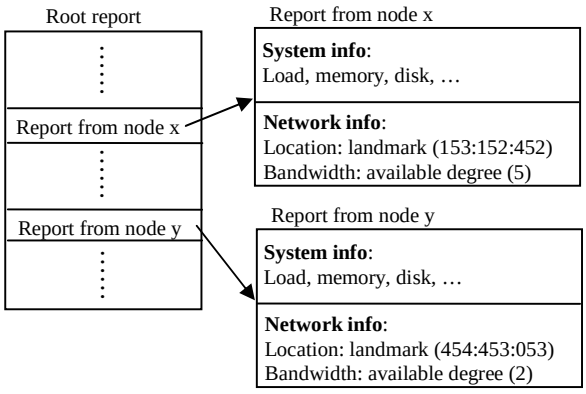


Figure 12: SOMO report structure for scheduling one single active ALM session.

Figure 12 gives an example of the SOMO root report and the detailed report submitted from individual nodes. Recall that each node (for instance node x and y) will continue to update through the SOMO hierarchy, resulting in continuous refreshing of the root report.

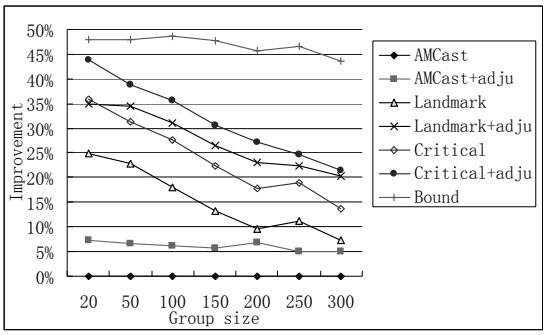


Figure 13: The performance of scheduling single ALM session. *AMCast* represents the original algorithm, *Critical* is our modified "critical node" heuristic, *Landmark* stands for the landmark based approach, *Bound* denotes the theoretical upper bound. *adju* denotes the combination when tree adjustment is performed.

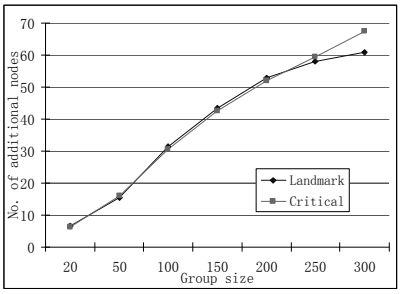
We are now ready to present our results. A fair evaluation should compare our results against those of a globally optimal algorithm. Since this is not available, we report our results in terms of percentage of tree height improvement relative to the AMCast algorithm. In other words, if H_{alg} is the tree height achieved using *alg*, then:

$$Improvement = (H_{AMCast} - H_{alg})/H_{AMCast}$$

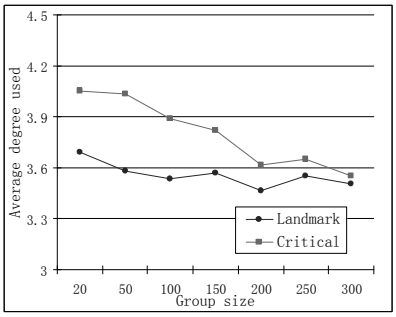
The upper bound is the latency between the furthest node to the root, corresponding to the ideal performance if the root has degree of infinity. For the data set that we used, the upper bound is between 40~50%. The average performance of these algorithms over 20 runs is shown in Figure 13 for various group sizes. It is conclusive that resource pool is very effective for small-to-medium group size. For larger groups, the original AMCast has more

rooms to optimize by using the existing members already. We believe that in reality, small groups are in fact more common.

For instance, *Landmark+adjustment*, which is a practical algorithm, delivers more than 30% latency reduction over the baseline for group size of 100; for group size of twenty, the reduction is 35%. Interestingly enough, tree adjustment, which is otherwise mediocre in shortening the tree (5% over baseline), is remarkably effective especially for *Landmark*. Part of the reason may be due to the inaccuracy introduced by using delay vectors to estimate node proximity.



(a)



(b)

Figure 14: (a) number of additional nodes used and (b) the average degree used on additional nodes.

It is also intriguing to see that using landmark instead of oracle the results are not substantially different. This demonstrates that our approach is practical. To understand that further, we compare the number of helper nodes and their average degrees in Figure 14, for *Critical* and the *Landmark*. It shows that both algorithms recruit about the same number of helper nodes. However, the quality of the nodes selected by the latter is not as high, especially when group size is small. This explains why the performance of *Critical* and *Landmark* converges for larger group (see Figure 13).

3.3 Scheduling Multiple ALM Sessions

The preceding section describes the stand-alone scheduling algorithm for one ALM session; we now discuss how multiple active sessions are scheduled in the system. Our goals are: 1) higher priority sessions are proportionally assigned with more resources, and 2) that the utilization of the resource pool as a whole is maximized.

All the sessions may start and end at random times. Each session has an integer valued priority between 1 and 3. Priority 1 session is the highest class. The number of maximum simultaneous sessions varies from 10 to 60 and each session has non-overlapping member set of size 20. Thus, when there are 60 active sessions, *all* nodes will belong to at least one session. That is, the fraction of *original* members of active sessions varies from 17% to 100%. Actual employed nodes will be greater by including helper nodes that lie outside the session members and, especially when such fraction is big, nodes with larger degrees may be involved in more than one session.

The principle underlying our approach is very simple, and it draws insight from a well-organized society: as long as global, on-time and trusted knowledge is available, it may be best to leave each task to compete resources with their own credentials (i.e., the priorities). Thus, we employ a hybrid model that combines global, on-time knowledge with individual, credential-based competition.

Setting the appropriate priorities at nodes involved in a session takes extra consideration. In a collaborative P2P environment, if a node needs to run a job which includes itself as a member, it is fair to have that job be of highest priority in that node. Therefore, for a session *s* with priority *L*, it has the highest priority (i.e. 1 in our experiment) for nodes in *M*(*s*), and *L* elsewhere (i.e., for any helper nodes lie outside *M*(*s*)). This ensures that each session can be run, with a lower bound corresponding to the *AMCast+adju* algorithm. The upper bound is obtained assuming *s* is the only session in the system (i.e., *Landmark+adju*).

As before, the root of an ALM session is the task manager, which performs the planning and scheduling of the tree topology. Each session uses the *Landmark+adjustment* algorithm to schedule completely on its own, based on system resource information provided by SOMO. For a session with priority *L*, any resources that are occupied by tasks with lower priorities than *L* are considered available for its use. Likewise, when an active session loses a resource in its current plan, it will need to perform scheduling again. Each session will also rerun scheduling periodically to examine if a better plan, using recently freed resources, is better than the current one and switch to it if so.

$d_{bound}(x)$	4
x.dt[1]	$2(S_4)$
x.dt[2]	0
x.dt[3]	$1(S_{1\,2})$

x's degree table

$d_{bound}(y)$	2
y.dt[1]	$2(S_5)$
y.dt[2]	0
y.dt[3]	0

y's degree table

Figure 15: two example degree tables.

To facilitate SOMO to gather and disseminate resource information so as to aid the planning of each task manager, each node publishes the following information in its report to SOMO:

- Its most recent measured delay vector to the landmarks. This part is the same as before.
- Its degree, broken down into priorities taken by active sessions. This is summarized in the *degree table*.

In Figure 15, we show the degree tables of two nodes. *x*'s total degree is 4, and is taken by session s_4 for 2 degrees, and s_{12} by another one degree, leaving *x* with one free degree. *y* on the other hand, has only two degrees and both of them are taken by session s_5. The degree tables are updated whenever the scheduling happens that affect a node's degree partition. Degree tables, as mentioned earlier, are gathered through SOMO and made available for any running task to query.

The original *AMCast+adjustment* is the base algorithm but is slightly extended so that resources are recruited from elsewhere with the guideline of priorities, and that the degree tables of nodes involved are appropriately updated. For completeness, the pseudo-code is listed in Figure 16. The procedure takes an integer *pri* as the session's priority. We now explain the changes:

- Line 10. Any helper node's degree is counted only for portions that are either free or occupied by lower priority tasks.
- Line 29-35: when the schedule is done, we set the degree tables of all the nodes in the plan appropriately: 1 (the highest priority) if they belong to the original member set, and *pri* otherwise. If other sessions are affected because their resources are taken away, their task managers are notified.

```
1.  ALM (r, V, pri) { //V==M(s), r is the root, pri is the priority
2.    for all v∈V                    // initialization
3.      height(v)=l(r, v); parent(v)=r
4.    T = (W={r}, Link={})
5.
6.    while (W<V) {                  // loop until finish
7.      find u∈{V-W} s.t. height(u) is minimum
8.      if (d(parent(u))==d_bound(parent(u)-1) { // find helper node
9.        find h in resource pool:   // adjust helper's degree
10.       d_bound(h) = d_bound(h) - sum(h.dt[i]), where i≤pri
11.       d_bound(h) ≥ 4  &&  l(h, parent(u))<R &&
12.       l(h, parent(u))+max(l(h, v) is minimum
13.       where v satisfies parent (v) = parent (u)
14.     }
15.
16.     if h≠NULL                    // integrate the helper node
17.       W+={h}; Link+={h, parent(u)};W+={u}; Link+={u, h};
18.     else
19.       W+={u}; Link+={u, parent(u)};
20.
21.     for all v∈{V-W} {            // re-adjust the height
22.       height(r)=∞
23.       for all w∈W {
24.         if d(w)<d_bound(w) && height(v)>height(w)+l(w, v)
25.           height(v)=height(w)+l(w, v); parent(v)=w
26.       }
27.     }
28.   adjust(T)
29.   for all v∈W {                  // record degree in the degree table
30.     if v∈V                       // v is in the original participant set
31.       v.dt[1] += d(v);
32.     else                         // v is a helper node
33.       v.dt[pri] += d(v);
34.     notify sessions, if any, whose resources are preempted
35.   }
36.   return T
37. }
```

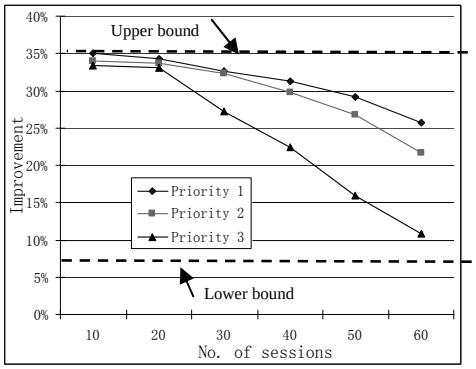
Figure 16: The algorithm to schedule one sessions when there are multiple active sessions. Changes are in line 10 and line 29-35.

Notification due to preemption is through the information recorded in the degree table. Re-planning is also run periodically when some resources are recently freed, in search for a better plan. To minimize the impact of rescheduling, each session is connected with a graph generated by the *AMCast+adju* as the backup plan.

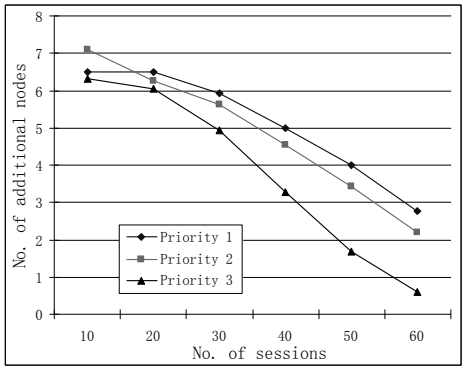
Ideally, the performance improvement should have a lower bound of *AMCast+adjust* where only the original member set is involved, and an upper bound of *Landmark+adjust*, when the session is the only active one in the resource pool. Therefore, performance will lie within 7%~35% reductions over *AMCast* (see data in Figure 13 when group size is 20).

The result is shown in Figure 17-(a). The x-axis is the number of active sessions, while the y-axis is the

performance improvement. To ease the comparison, the upper bound and lower bound are also shown.



(a)



(b)

Figure 17: (a) the performance of multiple ALM sessions and (b) the average number of additional nodes used.

As expected, the data perfectly drop into the interval between lower bound and upper bound. When there are more sessions and overall resource becomes scarce, performance decreases across the board. However, higher priority tasks are able to sustain much better than the lower ones, conforming to our predictions. Figure 17-(b) depicts the number of helper nodes taken, which shows that lower priority tasks lose more helper nodes when resource is under intense competition.

We also studied three other variations:

- When resource is already occupied by a running task, the current task will choose to skip this node and moves on, i.e., preempting is not allowed. The problem here is that some nodes' resources will be fully consumed even though they belong to the current task's member set. This will lead to more than 50% of sessions unable to form a tree.
- When the resource is occupied by a running task and it is not the current task's participant, then the current task will skip this node, otherwise preempting will occur. This turns out to be too conservative and higher priority tasks can hardly benefit from this approach. The reason being that earlier low priority sessions have occupied many good nodes which can be only preempted by their owners.
- Do not recycle those resources freed by a completed task to optimize existing, still active tasks. This is equivalent to artificially increasing the number of concurrent sessions. From Figure 17, we can see that the performance heavily depends on the number of live sessions. Therefore, this approach will make the performance worse unless sessions are relatively short.

Previous results are all based on a non-uniform distribution of degrees. This is reasonable since most of the clients are bandwidth starved. However, this may not be the case under some special circumstances such as corporate networks where lease lines are used. To understand the robustness of our algorithm as well as its sensitivity to other degree distribution, we ran another set of experiments where node degree is a random variable between 2 and 6, for group size of twenty. The performance bound in this case is [10%, 27%].

Figure 18 shows the result. It shows that the performance is quite robust in this case, i.e., it does not depend much on number of sessions, nor on priorities. This indicates that the resource competition is low. Therefore, a more conservative approach like the second and the third variation mentioned previously will be just as efficient, whereas overhead brought by rescheduling can be avoided. However, a session should still preempt other sessions that are running on their member set nodes, for otherwise more than 50% of the sessions will be unable to find valid solutions.

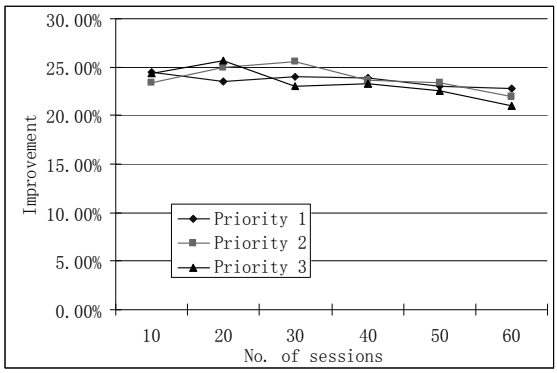


Figure 18: The performance of scheduling multiple ALM sessions under uniform degree distribution.

The above finding brought one interesting point, different scheduling algorithms might be necessary, depending on the degree distribution. Since through SOMO the degree distribution can always be discovered, adaptation of scheduling algorithm is possible.

3.4 Discussion

For a large and dynamic system, centralized resource scheduling will itself be a bottleneck, both in terms of scalability and stability. We believe that the right principle to adopt is to allow individual tasks to compete with their own credentials based on trusted global knowledge. The freshness of such global knowledge is bounded by O(log*N*) and, as a result, some of the properties in traditional single box system can be violated. For example, if two sessions s_1 and s_2 start close enough, their FIFO (First-in-first-out) property is hard, if not impossible, to enforce. Due to the distributed nature of job scheduling, the session starts later may reach (and thereby reserve) some resources earlier. However, resource allocation among tasks with different priorities can still be enforced – provided priority is authentic. What bounds and tradeoffs can such mechanism guarantee is an interesting future research topic.

4. RELATED WORK

Our work spans across a number of related fields: distributed data structure, scalable monitoring services, the concept of resource pool and its utilizations. We will discuss them in turn.

4.1 Self-Configured Monitoring Service

Data overlay relies on the key property of the P2P DHT that an item with unique key can be created and retrieved. In other words, DHT is a globally accessible and associative storage. In fact, the utilities of distributed hash table has been proposed earlier [7], but works such as Chord[16], Pastry[13], Tapestry[22] and CAN[10] emphasizes more on the self-organizing aspect. Data overlay has extended this property to arbitrary data structures.

A pure "peer-to-peer" mindset will view hierarchy as a forbidden word. We believe this is misleading as important functionalities such as aggregation and indexing [1][8] inherently imply a hierarchical structure. On this, SOMO bears the most similarity to Astrolabe [12], a peer-to-peer management and data mining system. SOMO operates at the rudimentary *data structure* level while Astrolabe is on a virtual, hierarchical *database*. SOMO's extensibility is much like that of active network, whereas Astrolabe uses SQL queries. The marked difference is that SOMO is designed specifically on top of P2P DHT, for two reasons: 1) we believe P2P DHT has established a foundation over which many other systems can be built and thus there is a need for a scalable resource management and monitoring infrastructure and 2) by leveraging P2P DHT (in fact, data overlay) the design and protocols of such infrastructure can be much simpler. In fact, one can envision the two be combined in interesting ways: a high level, expressive query language built over a scalable and structured middleware which is further layered on top of DHT. Distributed, in-network query processing has also been investigated in apparently un-related fields such as sensor network, though the emphasis there is quite different [9].

4.2 Resource pool and its utilization

Orchestrating a resource pool is a long-standing vision, especially in the Grid Computing arena [6]. Exploring heterogeneity lies therein has been articulated by [17]. SOMO provides a concrete example of how such resource pool can be realized.

Earlier work of ALM includes Aharoni's paper [2] and ESM [5]. Since then, quite a few other proposals and systems have emerged [3][14][15][21], including AMCast [15] from which our algorithm is derived. Researchers in P2P community quickly realized that application-level multicast maybe one of the showcases of P2P DHTs as well [4][11][23]. But both of these two approaches have some pitfalls: the first does not explore the potentials of a resource pool; and the second can not guarantee (for the time being) any QoS requirements, nor do they explore node heterogeneity. Given a resource pool, we have not only studied how to optimize one single ALM session, but also to schedule multiple simultaneous sessions with different priority levels, and have validated the hybrid model where global knowledge is combined with individual competition. To the best of our knowledge, this has been the first work along this line.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented our approach to optimize wide-area application-level multicasting in a collaborative resource pool. We construct the resource pool by combining P2P DHT's capability of self-organizing large amount of resources, and an in-system, efficient, scalable and fault-tolerant metadata aggregation infrastructure SOMO. Active ALM sessions are optimized by recruiting any spare resources nearby, and we have proved that practical algorithm can give substantial performance improvement. Our model combines global knowledge and individual, credential-based competition and is completely distributed, and is applicable to distributed job scheduling in a large resource pool in general.

We are currently building a wide-area testbed to test the idea of P2P resource pool, and the ALM scheduling algorithm is one of the experiments we plan to run.

6. REFERENCES

[1] Adamic, L., Huberman, B., Lukose, R., and Puniyani, *A. Search in Power Law Networks*, *Physical Review. E64*(2001), 46135-46143

[2] Aharoni E. and Cohen R. *Restricted Dynamic Steiner Trees for Scalable Multicast in Datagram Networks.* IEEE/ACM Trans. on Networking, Vol. 6, No. 3, Jun. 1998

[3] Banerjee S., Bhattacharjee B., etc. *Scalable Application Layer Multicast*. SigComm'02, Pittsburgh, USA, Aug. 2002

[4] Castro M., Druschel P., Kermarrec A., and Rowstron A. *SCRIBE: A Large-scale and Decentralized Application-level Multicast Infrastructure*. IEEE Journal on Selected Areas in Communications, Vol. 20. No 8. Oct. 2002

[5] Chu Y., Rao S., and Zhang H. *A Case for End System Multicast*. SigMetrics'00, CA, USA, Jun. 2000

[6] The Globus Project, http://www.globus.org

[7] Gribble et al. *The Ninja Architecture for Robust Internet-Scale Systems and Services.* In Journal of Computer Networks, Volume 35, Issue 4, March 2001.

[8] Lv, Qin, Ratnasamy, Sylvia and Shenker. Scott. *Can Heterogeneity Make Gnutella Scalable*? In IPTPS'02.

[9] Madden, S and et al. *TAG: A Tiny AGregation Service for Ad-Hoc Sensor Networks*. OSDI'02

[10] Ratnasamy, S., Francis P., Handley M., Karp R., and Shenker S. *A Scalable Content-Addressable Network.* SIGCOMM'01, San Diego, CA, USA, 2001

[11] Ratnasamy S., Handley M., Karp R., and Shenker S. *Application Level Multicast using Content Addressable Networks*, NGC'01, London, UK, Nov. 2001

[12] Renesse, V, Birman, R and Kenneth. *Scalable Management and Data Mining using Astrolabe.* In IPTPS'02.

[13] Rowstron A. and Druschel P. *Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems*. Middleware'01, Heidelberg, Germany, 2001

[14] Shi S. and Turner J. *Routing in Overlay Multicast Networks*. Infocom'02, New York, USA, Jun. 2002

[15] Shi. S., Turner J., and Waldvogel M. *Dimensioning Server Access Bandwidth and Multicast Routing in Overlay Networks*. NOSSDAV'01, New York, USA, Jun. 2001

[16] Stoica, I., Morris R., Karger D., etc. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. SIGCOMM'01, San Diego, CA, USA, 2001

[17] Teodosiu, D et al. Hardware Fault Containment in Scalable Shared-Memory Multiprocessors. In ISCA'97.

[18] Zegura E., Calvert K., and Bhattacharjee S. *How to Model an Internet-work*. InfoCom'96 , CA, USA, May 1996

[19] Zhang B., Jamin S., and Zhang L. *Host Multicast: A Framework for Delivering Multicast to End Users.* Infocom'02, New York, USA, Jun. 2002

[20] Zhang Z., Shi S., and Zhu J. *SOMO: Self-Organized Metadata Overlay for Resource Management in P2P DHT*. In IPTPS'03.

[21] Zhao, B., and et al. *Brocade, Landmark Routing on Overlay Networks*. In IPTPS'02.

[22] Zhao B.Y., Kubiatowicz J.D., and Josep A.D. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, 2001

[23] Zhuang S.Q., Zhao B.Y., and Joseph A.D. *Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination*, NOSSDAV'01, New York, USA