Resource Discovery Techniques in Distributed Desktop Grid Environments

Jik-Soo Kim, Beomseok Nam, Peter Keleher, Michael Marsh, Bobby Bhattacharjee and Alan Sussman
UMIACS and Department of Computer Science
University of Maryland at College Park
{jiksoo, bsnam, keleher, mmarsh, bobby, als}@cs.umd.edu

***Abstract*— Desktop grids use opportunistic sharing to exploit large collections of personal computers and workstations across the Internet, achieving tremendous computing power at low cost. Traditional desktop grid systems are typically based on a client-server architecture, which has inherent shortcomings with respect to robustness, reliability and scalability. In this paper, we propose a decentralized, robust, highly available, and scalable infrastructure to match incoming jobs to available resources. Through a comparative analysis on the experimental results obtained via simulation of three different types of matchmaking algorithms under different workload scenarios, we show the trade-offs between efficient matchmaking and good load balancing in a fully decentralized, heterogeneous computational environment.**

I. Introduction

Desktop grid computing has achieved tremendous computing power with low cost through opportunistic sharing to exploit large collections of personal computers and workstations across the Internet. Existing platforms for desktop grid computing typically employ a client-server architecture, where a trusted server supplies jobs to a set of potentially unreliable client machines [1], [2]. This architecture has inherent shortcomings with respect to robustness, reliability and scalability.

Our goal is to design and build a massively scalable infrastructure for executing grid applications on a widely distributed set of resources. Such infrastructure must be *decentralized*, *robust*, *highly available* and *scalable*, while effectively *mapping* application instances to available resources throughout the system. By employing Peer-to-Peer (P2P) services, our techniques allow users to submit jobs to the system, and the jobs to be run on any available resources in the system that meet the minimum job requirements (e.g., memory size, disk space, etc.). The overall system, from the point of view of a user, can be regarded as a combination of a centralized, Condor-like Grid system for submitting and running arbitrary jobs [3], and a system such as BOINC [1] for farming out jobs from a server to be run on a potentially very large collection of machines in a completely distributed environment. Such a confluence of P2P and distributed computing is a natural step in the progression of grid computing, and has indeed been described as inevitable [4], [**?**], [5].

Our preliminary work [6] has shown that we can effectively match jobs to processing nodes with varying capabilities by leveraging routing information from an underlying P2P system, and by efficiently aggregating and disseminating resource utilization information. However, as such a system scales to large configurations and heavy workloads it becomes a challenging problem to efficiently match jobs with different resource requirements to available heterogeneous computational resources, to provide good load balancing, and to obtain high system throughput and low job turnaround times.

In this paper, we quantify the trade-offs between performing efficient matchmaking and maintaining good load balance, comparing three different matchmaking algorithms for several different types of workloads via simulation. This study is intended to give insight into the design and implementation of resource discovery algorithms in a distributed and heterogeneous Grid environment.

The rest of the paper is structured as follows. Section II presents related work. Section III discusses our assumed context and overall goals, while Section IV describes the algorithms and optimization criteria for matching jobs to resources. Finally, Section V contains our evaluation, and Section VI concludes.

II. Related Work

Peer-to-Peer research has shown that a robust, reliable system for storing and retrieving files can be built upon unreliable machines and networks. The algorithms for object location and routing in P2P networks [7], [8], [9], [10] are capable of scaling to very large number of peers and simultaneous requests for service (called *Distributed Hash Tables* or DHTs). Building upon these basic services to provide a system for making computational resources available on demand can allow users to both provide resources when they are not being otherwise used, and to obtain resources when they are needed.

Research such as [**?**], [11] proposed a P2P architecture to locate and allocate resources in Grid environment employing a *Time-To-Live* (TTL) mechanism. TTL-based mechanisms are relatively simple but effective ways to find a resource (that meets the job constraints) in a widely distributed environment without incurring too much overhead in the search. However, such mechanisms may fail to find an appropriate resource to run a given job on (that meets the job constraints), even though such a resource exists somewhere in the network, because of the TTL mechanism. Therefore, to *always* find an appropriate resource in the system (if it exists) without much overhead, we must employ more effective algorithms, as described in Section IV.

Studies on encoding static or dynamic information about computational resources using a DHT hash function for resource discovery have also been conducted [4], [12]. In particular, the SWORD system [12] explored a variety of architectures, including a centralized data center, P2P based resource discovery and hybrid architectures. However, there can be a load balancing problem for these encoding techniques, since a small fraction of the nodes can contain a majority of the resource information whenever there are many nodes that have very similar (or identical) resource capabilities in the system.

The CCOF (Cluster Computing on the Fly) project [13] conducted a comprehensive study of generic searching methods in a highly dynamic P2P environment to locate idle computer cycles throughout the Internet. More recent work from the CCOF researchers, on a peer-based desktop grid system called WaveGrid, constructed a *timezone-aware* overlay network based on Content-Addressable Network (CAN) [7] to use idle night-time cycles geographically distributed across the globe [14]. However, the host availability model in that work is not based on the resource requirements of the jobs and that work does not consider balancing load across the available system resources.

Awan et al. [15] proposed a distributed cycle sharing system that utilizes a large number of participating nodes to achieve robustness through redundancy on top of an unstructured P2P network. By employing efficient uniform random sampling using random walks, probabilistic guarantees on the performance of the system could be achieved. Also, they claim to support robustness and scalability with high probabilistic guarantees. However, as for the CCOF project, the job allocation model in this work does not consider the constraints of the jobs nor the varying resource capabilities of nodes in the system.

III. Workload Assumptions and Overall Goals

A general-purpose system must accommodate heterogeneous clusters of nodes running heterogeneous batches of jobs. The obvious implication is that a matchmaking process must incorporate both node and job information into the process that eventually maps a job onto a specific node.

Our expected environment and usage simultaneously makes this problem easier and more difficult. A large fraction of nodes in our system might belong to one of a small number of equivalence classes. For example, many organizations buy clusters of identical machines all at once, whether to create compute farms or just to replace an entire department's machines at once. Node clusters make the problem more difficult by removing the notion of a single best match for a given job. The underlying routing algorithm must be able to cope with many similar nodes and perform some intelligent load balancing across them. However, node clustering can also simplify the problem by reducing the set of possible choices for the routing algorithm.

Likewise, job mixtures might show clustering. Sets of similar jobs (in terms of resource constraints) can result from running the same code with slightly different input datasets. For example, users often perform *parameter sweeps* to optimize algorithmic settings or explore the behavior of physical systems. Similarly, the same computation may be performed on different input regions, such as n-body or weather calculations that differ only in spatial coordinates.

To summarize, the goals of any matchmaking (or in our case, routing) algorithm must include the following:

1) *low overhead* - The routing must not add significant overhead to the cost of executing a job. This can be challenging, given that the routing/matching is done in a completely decentralized fashion.
2) *completeness* - A valid assignment of a job to a node must be found if such an assignment exists.
3) *precision* - Resources should not be wasted. All other issues being equivalent, a job should not be assigned to a node that is over-provisioned with respect to that job, such that the over-provisioning does not give the job an advantage.
4) *load balance* - Load (jobs) must be distributed across the nodes capable of performing them.

There are additional issues that we do not discuss here. For example, in some situations (e.g., conditions of low load), the system might prefer to optimize throughput by executing jobs on the *most* capable available node. This raises the question of what we wish to optimize for: throughput or response time. We are explicitly avoiding this issue by designing an infrastructure that can accommodate either objective.

IV. Algorithms

We begin by defining terminology and the basic framework of our approach to matchmaking, and then describes the two approaches that we evaluate in this paper: the *rendezvous node tree*, and *CAN-based resource matching*.

A. Terminology and Basic Framework

All of the work described here assumes an underlying distributed hash table (DHT) infrastructure [7], [8], [9], [10]. DHTs use computationally secure hashes to map arbitrary identifiers to random nodes in a system. This randomized mapping allows DHTs to present a simple insertion and lookup API that is highly robust, scalable, and efficient. We insert both nodes and jobs into a single DHT, performing matchmaking by mapping a job to a node via the insertion process, and then relying on that node to find candidates that are able and willing to execute the job. By leveraging such an architecture, we are effectively reformulating the problem of matchmaking to one of routing, similarly to anycasting [16], or content-based routing [17]. Jobs are injected into the system by *forwarding* them to a node that will become responsible for them.

A *job* in our system is the data and associated profile that describes a computation to be performed. A job profile contains several characteristics about the job, such as the client that submitted it, its minimum resource requirements, the location of input data, etc. All jobs in our system are independent, which implies that no communication is needed between them. This is a typical scenario in a desktop grid

environment, enabling many independent users to submit their jobs to a collection of node resources in the system.

Clients insert jobs into the system by submitting them to any system node. Nodes receiving submitted jobs assign them globally unique identifiers (GUIDs), and initiate the process of assigning them to *owners*.

An owner is responsible for monitoring the execution of the job and ensuring that its results are returned to the client. The owner attempts to find an appropriate *run node* through a matchmaking mechanism. Matchmaking is the process of matching jobs with physical resources, and consists of finding an appropriate node for running a job based on the constraints in the job profile and the current (distributed) state of the nodes in the system. Once an appropriate run node is identified, the new job is inserted into its incoming job queue where jobs are executed in FIFO order.

Run nodes periodically send *heartbeat* messages to the owners of all jobs either running or queued locally. Heartbeats are communicated directly between run nodes and owner nodes, rather than through DHT routing. This soft-state message plays an important role in failure recovery during the processing of jobs in our system, as job profiles are replicated on both the owner and run nodes. If either the owner node or the run node fails, the other will detect the failure and initiate a recovery protocol so that the job can continue to make progress. If both fail before the recovery protocol completes, the client must resubmit the job. After a job completes, the run node returns the results to the owner, which forwards them to the client.

B. The Rendezvous Node Tree

DHTs provide robustness, availability, and above all, scalability. They also introduce randomness into the system by mapping names to nodes through hash functions. This randomness helps balance routing load in DHTs, but we can also use it to help balance computational load in our desktop grid. For example, a crude form of load balancing can be accomplished merely by randomly choosing a node assignment from all viable candidates. The main drawbacks of this approach are that it does not account for dynamic aspects, such as the load at individual nodes in the system at any given time, nor does it describe a way to make a match when a randomized hashing matches a job with a node that is not capable of performing it ("completeness").

We begin with a description of the *rendezvous node tree* (RN-Tree or RNT), an approach to addressing both problems through use of a distributed data structure built on top of an underlying DHT, which in our implementation is Chord [9]. Specifically, the RN-Tree copes with dynamic load balance issues by performing a limited random walk after the initial mapping, and addresses completeness by passing information describing the most capable reachable system up and down the tree. This latter aspect allows even the hardest corner cases to be satisfied in $O(\log N)$ additional steps.

An RN-Tree contains all participating nodes in the desktop grid. Each node determines its parent node based on only local information, which enables building the tree in a completely decentralized manner (to find the parent node in the RN-Tree, divide the GUID of the predecessor node of the child node in the Chord ring by two and find the *successor* node of that GUID in the Chord ring - see [6] for more details). Since the GUIDs of nodes in the system are generated uniformly at random, the overall height of the RN-Tree is likely to be $O(\log N)$ where *N* is the total number of live nodes in the system (we investigated the characteristics of the RN-Tree in terms of overall height and node degree in [6]). Due to the dynamics of the system (new nodes joining, existing nodes departing), the correct parent pointer of a node can change over time. Therefore each node must refresh/update its RN-Tree parent node pointer periodically to maintain the RN-Tree structure.

Once the parent-child relationship in the RN-Tree is determined, each node periodically sends local subtree resource information (for the subtree rooted by that node) to its parent node, and this information is *aggregated* at each level of the RN-Tree (hierarchical aggregation as in [18]).

In the work described in this paper, the only information distributed through the tree is a description of the maximal amount of each resource available at some node in the subtree. The resources modeled include continuous variables, such as the speed of the CPU, the amount of memory available, and the amount of disk space available, and discrete variables such as operating system type and version. The resources modeled match the constraints (requirements) that can be specified in job profiles.

We inject jobs into the system by mapping each to a randomly-chosen node, which becomes the job's owner. The owner initiates a search for a node on which to run the job. The search first proceeds through the subtree rooted at the owner, only searching up the tree into subtrees rooted at the ancestors of the owner if the subtree does not contain any satisfactory candidates. The search is pruned using the maximal resource information carried by the RN-Tree.

Rather than stopping at the first candidate capable of executing a given job, the search proceeds until at least k capable nodes are found. The search completes by choosing the *least loaded* of the k nodes to run the job. To determine the least loaded node among the candidate run nodes, we poll each candidate for the *size of its job queue* (the current set of unfinished jobs assigned to a node) at the time the matchmaking is performed. Queue size is modeled as either the number of jobs in the queue (which was used in the experiments) or an estimate of the run time for all current jobs in the queue. Through experiments not discussed here, we have determined that a value of five (5) for k produces robust results with low overheads. Further details about this search procedure can be found in [6].

C. Content-Addressable Network

A content-addressable network (CAN) is a DHT that maps GUIDs to points in a d-dimensional space [7]. The conventional use of CAN is to map a GUID into the space by applying d different hashes, one for each dimension. However, positions

in the CAN space need not be created through randomized hashes. For example, Tang et al. [19] map documents and queries into a CAN space, executing queries via a blind local search centered on a query's mapping.

Similarly, we can formulate our matchmaking problem as a routing problem in a CAN space. By treating each resource type as a distinct dimension, nodes and jobs can be mapped into the CAN space by using their capabilities or constraints on each resource type to determine their coordinates. As a simple example, if our resource types consist of CPU speed, memory size, and disk space, we might map a 3.6GHz workstation, with 2GB of memory and 500GB of disk space, to the point $\{360, 2000, 500\}$. A job requiring at least a 1GHz machine, 100MB of memory, and 200 MB of disk space would map to $\{100, 100, 0.2\}$, clearly some distance from the node discussed above. With this approach, mapping a job to a node might seem to consist merely of mapping the job into the CAN space and finding the nearest node.

However, the semantics of matching jobs to nodes are different than that of merely finding the closest match node. Most importantly, job constraints represent *minimum* acceptable quantities. Any node meeting a job's constraints can run the job, but a node whose coordinate in any dimension is less than that specified by the job's constraints, even if very close in the CAN space, is not a viable choice to run the job. Hence, instead of searching for the node whose capabilities are closest to the job's constraints, our matchmaking/routing procedure must search for *the closest node whose coordinates in all dimensions meet or exceed the job's constraints*.

A second issue is that jobs might not have constraints in all dimensions. Indeed, a job may be injected into the system with no constraints at all, implying that it may be executed by any node in the system. We map any undefined constraint to the minimum in the corresponding dimension. This approach is simple and correct, but might exacerbate load balance problems. We discuss this issue more in Section VI.

1) Details: A CAN works by dividing the d-dimensional volume into zones managed by individual nodes. Zone assignment is accomplished by mapping new nodes to an existing zone, and then giving the new node part of that zone. Zones are re-assigned and aggregated when existing nodes leave or fail. These techniques can be used to divide the CAN among distinct run nodes. However, mapping of jobs to those nodes is less straightforward.

A job is inserted into the system by using its constraints as coordinates, and defining the owner of the resulting zone as the owner of the job. The owner creates a list of candidate run nodes, and chooses the least loaded among them at the time matchmaking is performed (as reported by the candidate nodes), as for the RN-Tree algorithm. The candidate nodes are drawn from the owners of neighboring zones, such that each candidate is at least as capable as the original owner in all dimensions (capabilities), but more capable in at least one. As with the RN-Tree mechanism, we used sensitivity analysis to identify five (5) as a reasonably robust constant size for this set. Owner nodes store information about neighbors, so an owner may be able to create a candidate list locally, without any communication.

2) Virtual Dimensions: The above procedure works in all cases, but may cause extreme load imbalance when many nodes have similar, or even identical, resource capabilities. Since the coordinates of a node are defined by its resource capabilities, identical nodes are mapped to the same place in the CAN volume. The best way distribute ownership of a zone across multiple such nodes is not immediately obvious.

Conversely, many jobs might have extremely similar constraints. For example, many jobs will probably be inserted into the system with no constraints at all. In this case, all of the these jobs will be mapped to a single node that owns the zone containing the minimum point in the CAN volume.

We address this issue by supplementing the "real" dimensions (those corresponding to node capabilities) with a *virtual dimension*. Coordinates in the virtual dimension are generated uniformly at random. Whenever a new node joins the system, a representative point for the new node is generated by combining the resource capabilities of the node and randomly generated virtual dimension values. Therefore, even when multiple identical nodes join the system, they are mapped to distinct locations, and zone splitting is straightforward. Similarly, when a new job is inserted into the system, the new job's coordinates become a combination of the job's constraints and a randomly assigned virtual coordinate. In combination, the randomly assigned node and job coordinates act to break up clusters and spread load more evenly over nodes.

3) Changes to Underlying CAN: Our use of CAN differs from the canonical uses in that coordinates have semantic meaning. This difference requires several changes in how the underlying network management algorithms work. The most important changes are in the way zones are split and merged.

Zones are split when a new node enters the system. The CAN maps the node to an existing zone, and then the zone is split between the owner and the new node. The default CAN split algorithm can choose to split the zone on any axis, because the mapping of a zone to an owner has no semantics, and the coordinates of a pair of points usually differ on most, if not all, axes. In our CAN, however, nodes and jobs may be identical in capabilities and constraints, differing only in their coordinates in the virtual dimension (e.g. for a cluster of homogeneous nodes, since we use the resource capabilities as the representative point for each node in the system). This restricts the choice of the dimension on which to split. Therefore, our split mechanism first tries to find a split axis among the real dimensions that have different coordinates across the existing node and the new node. If that is not possible, the virtual dimension is used as the split axis. To build a better (i.e. closer to cubic) grid space when splitting real dimensions, we iterate across the dimensions for each split operation.

The second major change to the algorithms is in how zones are merged. A zone is merged with a neighbor when it is orphaned because of an owner leaving, either gracefully or by

failing. The default CAN recovery algorithms allow such an orphaned zone to be merged with any neighboring zone. No restriction is made on which nodes can own a zone. In fact, a node can own multiple zones, which can result in highly fragmented coordinate space. Therefore, to achieve a one-to-one node to zone assignment, CAN runs a periodic *background zone reassignment* algorithm. That algorithm can assign one of the neighbor nodes of the departed node to another region, without any restrictions on merging and reassigning the orphaned zone (see the details in [7]). However, in our system this can cause a violation in our required semantics about the relationship between a zone and the owner of that zone, whereby a zone should contain the coordinates (i.e., resource capabilities) of its owner.

Specifically, zone owners play two roles. First, they ensure that jobs mapped to the zone are run. This is accomplished by creating a set of candidate run nodes and polling them to find the least loaded candidate run node. For this purpose, the owner of a zone would not actually have to be mapped into that zone, because a job's owner is never a candidate to run the job. However, owners also serve as candidate run nodes for jobs mapped to neighboring zones. For example, assume a job is mapped into a zone z_i, and that zone z_j is z_i's neighbor. z_i's owner may then include z_j's owner in the list of candidate run nodes for any job mapped to z_i. However, if z_j's owner is not actually mapped somewhere in z_j, it might not have the capabilities z_i's owner expects, and might therefore not be able to run the job. The zone merging procedure must therefore preserve the constraint that a zone's owner must be mapped into the zone. Satisfying this constraint requires that zones be merged in a way that is consistent with the original split order. The zone merge algorithm accomplishes this by preserving the original split order at the owner, and reversing that order to select which node to merge a zone into.

D. Centralized Matchmaker

To compare against the RN-Tree and CAN-based matchmaking algorithms, we have designed an *online* scheduling mechanism, called the *Centralized Matchmaker*, that maintains global information about the current capabilities and load information for all the nodes in the system, so can assign a job to the node that both satisfies the job constraints and has the minimum job queue size across all nodes in the entire system (breaking ties arbitrarily). In our simulation environment, the Centralized Matchmaker does not incur any cost for gathering the global information about the nodes in the system and performing the matchmaking (since the simulator can maintain global information about all the nodes in the system). Even though the matchmaking performed by the Centralized Matchmaker is not always optimal (since it is an online algorithm), it should provide good load balancing and can be a good comparison model for other matchmaking algorithms, as in [12], [13].

We can view the Centralized Matchmaker algorithm as the extreme case of the RN-Tree or CAN based search algorithm, since it first finds *all* candidate run nodes that meet the job constraints and picks the one with the shortest job queue. However, such a scheme would not be feasible in a complete system implementation with respect to performance and robustness, since the algorithm would incur a large overhead to find *all* nodes in the P2P system that meet the job constraints, and the node performing the centralized algorithm would be a single point of failure in the system.

V. Performance

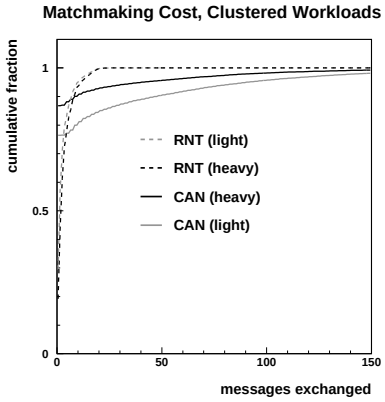
We present a preliminary evaluation of our ideas on decentralized job assignment.

A. Experimental Setup

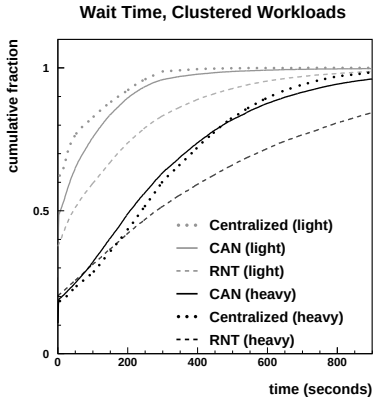
We use synthetic job and node mixes to simulate the behavior and measure the performance of both the RN-Tree and CAN-based approaches. Our intent is to model a P2P desktop grid environment with a heterogeneous set of nodes and jobs. We therefore generated a variety of workloads, each describing a set of nodes and events. Events include node joins, departures (graceful or otherwise), and job submissions. The events are generated using a Poisson distribution with an arrival rate of 1 / [Average Event Inter-Arrival Time (AEIAT)]. Jobs can specify constraints for three different resource types: CPU speed, memory, and disk space. We generated node profiles using a clustering model to emulate resources available in a heterogeneous environment, where a high percentage of nodes have relatively small values for their available resources and a small fraction of nodes have larger amounts of available resources (as in [14]).

Though we investigated many workloads, we have space in this paper only for the most interesting results. The first four workloads are relatively static; no nodes join or leave during the course of the experiments. They differ on two axes. Workloads are categorized as either *clustered* or *mixed*. The former divides all nodes and jobs into a small number of equivalence classes, where all items in a given equivalence class are identical. The latter assigns node capabilities and job constraints randomly. Workloads are also distinguished by whether the jobs have "light" or "heavy" constraints. For a given job, each type of resource has a fixed independent probability of being constrained: "light" jobs have an average of 1.2 constraints (out of the 3) and "heavy" jobs have an average of 2.4. More detail on the workload generation can be found in Kim et al. [6].

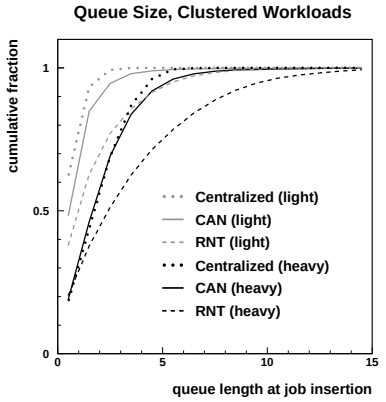
Our metrics are *matchmaking cost* (the number of messages required for finding candidate run nodes by the owner node of a job), *wait time* (the amount of time between when a job is injected and when it actually starts running), and *average queue length*, which is the length of the non-preemptive run queue seen by a job when it is finally assigned to a run node. Matchmaking cost directly quantifies the messaging cost needed to perform the matchmaking in a decentralized manner. Wait time includes the time to perform the matchmaking algorithm *and* the time spent waiting in the run queue before a job is performed. Wait time reflects both protocol overhead and the quality of the matchmaking results, i.e., load imbalance.



(a) Matchmaking Cost

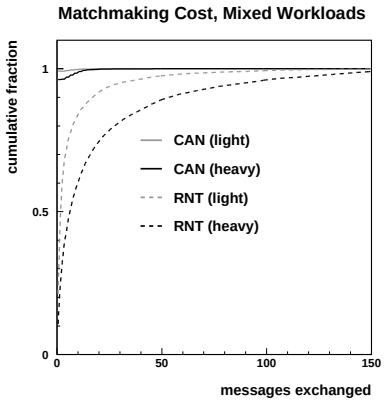


(b) Wait Time

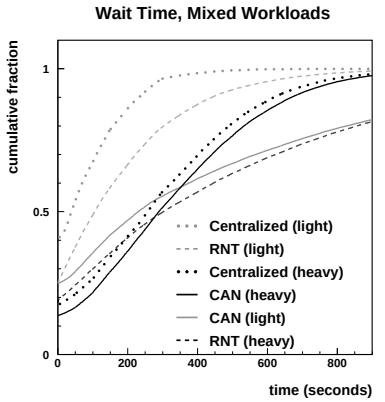


(c) Queue Length

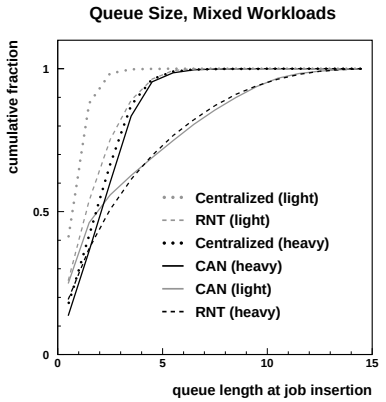
Fig. 1. Clustered Workloads



(a) Matchmaking Cost



(b) Wait Time



(c) Queue Length

Fig. 2. Mixed Workloads

Finally, the distribution of queue lengths provides a direct measurement of the load balance seen by injected jobs.

We test the RN-Tree approach (RNT), the CAN approach (CAN), and the idealized centralized approach (Centralized) that uses up-to-date global information to choose the node with shortest queue length from all nodes in the system. We do not include "matchmaking cost" numbers for the centralized approach because it requires no messages.

Overall, we found that, as expected, the rendezvous approach is better suited for less demanding situations with more structure. The CAN approach is generally better suited to handle more complex situations. However, we did encounter a few surprises.

B. Performance

Figure 1 shows "matchmaking cost" (in messages), "wait time", and "queue length" for the clustered workloads, while Figure 2 shows the corresponding data for the mixed workloads. Taking the clustered workloads first, the RN-Tree has lower matchmaking costs, but CAN has lower wait times and smaller queue lengths. The difference in queue lengths explains the difference in wait times, and comes about because the virtual dimension allows nodes of a cluster to be spread in the CAN space.

The mixed workloads give us a slightly different story. The matchmaking costs (number of messages) and the wait time on the "heavy" constraint workload still favor CAN, but CAN's performance on the "light" constraint mixed workload is much worse than that of RNT. Figure 2(c) shows that queue lengths are much larger and more varied in CAN than RNT, implying load imbalance.

This latter finding was somewhat of a surprise: CAN performs poorly with the "light" mixed workload. To understand why the resulting load imbalance is worse than in the clustered case, consider a hypothetical CAN with only a single real dimension, CPU speed, and where each node has a CPU value of "3600", and each job has a constraint of "2000". Nodes and jobs end up distributed along two parallel lines in the 2-D CAN because of their randomly assigned virtual coordinates. As a result, the "closest" node can be different for each job. This is an approximation of the clustered case, and explains why CAN's load balance is good.

However, the analogue for the "light" mixed workload in this example would be slightly different. Since the workload is "light", most jobs would not have CPU constraints, causing their CPU speed coordinates to be the minimum value in that dimension, say, "1000". Hence, the jobs are still mostly

distributed (via the virtual dimension) along a line at a single CPU coordinate, much like the case above. However, the fact that this is the mixed workload implies that most nodes have distinct CPU speeds. The node with the lowest CPU speed ends up being closest to the line representing the jobs, and will become the owner of a disproportionate number of them, resulting in load imbalance.

Due to these characteristics of space partitioning for the CAN approach, the matchmaking performance of CAN shows different behavior for the clustered and mixed workloads. In particular, for clustered workloads the matchmaking cost for CAN is higher than that for RN-Tree. In the clustered workloads, many nodes have identical resource capabilities so that overall the CAN space will be split along the virtual dimensions. This results in *coarse-grained* ranges in the real dimensions, where each node maintains much larger zones compared to its own resource capabilities. Therefore, the matchmaking process in CAN becomes expensive for jobs that have a small number of very high resource requirements. However, for jobs that have more constraints, the overall matchmaking performance is better since jobs with many constraints are more likely mapped to the right region in the space where many candidate run nodes are available. On the other hand, for the mixed workloads, since there are not too many identical nodes in the system, the CAN space is partitioned along the real dimensions so that CAN clearly outperforms RNT in terms of matchmaking cost.

Figure 3 shows average wait times for three "light" mixed workloads where between 10% and 30% of the nodes leave during the course of simulation. Node departures are evenly split between graceful departures, where a node informs its neighbors before leaving, and failures, where the neighbors learn of the departure from the failure of heartbeat messages. Like the "light" mixed workloads discussed above, the CAN approach has problems balancing load. Both approaches perform poorly relative to the centralized approach because of the need to recover and reconfigure the network. This disparity grows with increasing departure rates. CAN's performance appears to be more affected than RNT's by the increasing departure rates. We speculate that the culprit is the much more complex recovery process, which involves zone reclamation and re-aggregation.

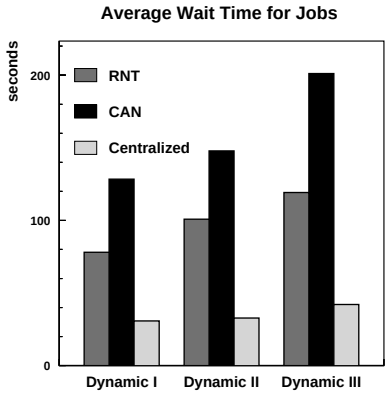


Fig. 3. Dynamic Workloads

VI. Conclusions

In this paper we have described two approaches to using P2P protocols in providing job scheduling and resource matching facilities to desktop grids. Overall, the CAN algorithm appears to produce significantly lower wait times than the RN-Tree approach over a broader spectrum of input. However, the RN-Tree and CAN approaches have different underlying rationales. The idea motivating the RN-Tree approach is to balance load by randomizing job assignment, mitigating the cost of matching demanding jobs by passing static capacity information around the tree. Job assignment essentially consists of a randomized mapping, followed by a short random walk to find a lightly-loaded node. The idea behind the CAN approach is to first find a node whose capabilities approximately match the job's constraints, followed by a short random walk among similar nodes to find one that is lightly loaded.

Our original expectation was that the overhead of the simpler RN-Tree protocol would be significantly less than that of CAN, but produce worse assignments. However, the RN-Tree's advantage in protocol cost turned out to be less than expected. Even relatively serious concerns, such as the cost of CAN's failure recovery process, can probably be addressed through techniques such as virtualization of the entire d-dimensional volume. Further, in all cases, protocol cost turned out to be less significant than wait times caused by load imbalance.

Both RN-Tree and the CAN approach can cause poor load balance in at least two ways. First, the random walks may not be long enough to find existing lightly-loaded nodes. This occurs because neither search is guided by dynamic load information. However, it may be less serious for the CAN approach because each CAN node stores a limited amount of load information for neighbor nodes.

A second potential cause of load imbalance is poor matches between jobs and nodes (i.e, poor *precision*). RN-Tree might be thought of as a "first-fit" algorithm; it selects as the run node the most lightly loaded of a set of randomly chosen nodes, such that each node meets the minimum job constraints. However, the chosen run node might be greatly over-provisioned for the job, and this over-provisioning might not be useful. For example, over-provisioning in terms of CPU rate may be useful because it can speed the execution of a given job, but an extra GByte of memory might not help execution time, and therefore not be useful. Meanwhile, other jobs needing the extra memory might be needlessly queued. By contrast, CAN is more of a "best-fit" algorithm (more precise) because the search starts at the node most closely matching the job's constraints.

The result is that the CAN approach is both more flexible and more efficient for the general case where the workload has a great deal of diversity. However, CAN's poor performance with the "light" mixed workload is an indicative of a broader problem in the robustness of the load balancing. While the

virtual dimension helps to smooth clumpy job and node distributions, thereby enabling better matchmaking, it is not sufficient in cases such as those described above.

One approach to improving load balance in these cases might be to add random increments to all dimensions, effectively "virtualizing" each of the real dimensions to a limited extent. Random increments would help spread out both jobs and nodes when the workload contains significant structure, or clustering. The difficulty lies in determining the proper amount of randomness to add to the system. Too little and load balance is fragile; too much and matchmaking becomes less precise, also adding load imbalance. The best approach is probably one that adapts dynamically both to the workload, and to current queue length distributions. However, both of these properties are global, and our approach is to make all decisions locally, in as decentralized a fashion as possible. The reconciliation of these competing concerns into a single protocol is the main focus of our ongoing research.

Finally, we plan to build and deploy a prototype system to look at real-world issues arising from heterogeneous environments running real applications. With help from our application-area collaborators in physics and astronomy, we will measure and report on the behavior of our system for real workloads on real systems.

REFERENCES

[1] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Nov. 2004.

[2] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 597–610, May 2003.

[3] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[4] A. S. Cheema, M. Muhammad, and I. Gupta, "Peer-to-peer Discovery of Computational Resources for Grid Applications," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID 2005)*, Nov. 2005.

[5] J. Ledlie, J. Schneidman, M. Seltzer, and J. Huth, "Scooped, Again," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[6] J.-S. Kim, B. Bhattacharjee, P. J. Keleher, and A. Sussman, "Matching Jobs to Resources in Distributed Desktop Grid Environments," University of Maryland, Department of Computer Science and UMIACS, Tech. Rep. CS-TR-4791 and UMIACS-TR-2006-15, Apr. 2006.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in *Proceedings of the ACM SIGCOMM*, Aug. 2001.

[8] A. Rowstran and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Nov. 2001.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proceedings of the ACM SIGCOMM*, Aug. 2001.

[10] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, Jan. 2004.

[11] A. R. Butt, X. Fang, Y. C. Hu, and S. Midkiff, "Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing," in *Proceedings of the 3rd Virtual Machines Research and Technology Symposium (VM'04)*, May 2004.

[12] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Design and Implementation Tradeoffs for Wide-Area Resource Discovery," in *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[13] D. Zhou and V. Lo, "Cluster Computing on the Fly: Resource Discovery in a Cycle Sharing Peer-to-Peer System," in *Proceedings of the 4th International Workshop on Global and Peer-to-Peer Computing*, Apr. 2004.

[14] ——, "WaveGrid: a Scalable Fast-turnaround Heterogeneous Peer-based Desktop Grid System," in *Proceedings of the 20th International Parallel & Distributed Processing Symposium*, Apr. 2006.

[15] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama, "Unstructured Peer-to-Peer Networks for Sharing Processor Cycles," *Parallel Computing*, vol. 32, no. 2, Feb. 2006.

[16] C. Partridge, T. Mendez, and W. Milliken, "Host anycasting service," Internet Engineering Task Force, Request for Comments 1546, Nov. 1993.

[17] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Dec. 1999.

[18] P. Yalagandula and M. Dahlin, "A Scalable Distributed Information Management System," in *Proceedings of the ACM SIGCOMM*, Aug. 2004.

[19] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks," in *Proceedings of the ACM SIGCOMM*, Aug. 2003.