Using Content-Addressable Networks for Load Balancing in Desktop Grids [*]

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

ABSTRACT

Desktop grids have evolved to combine Peer-to-Peer and Grid computing techniques to improve the robustness, reliability and scalability of job execution infrastructures. However, efficiently matching incoming jobs to available system resources and achieving good load balance in a fully decentralized and heterogeneous computing environment is a challenging problem. In this paper, we extend our prior work with a new decentralized algorithm for maintaining approximate global load information, and a job pushing mechanism that uses the global information to push jobs towards underutilized portions of the system. The resulting system more effectively balances load and improves overall system throughput. Through a comparative analysis of experimental results across different system configurations and job profiles, performed via simulation, we show that our system can reliably execute Grid applications on a distributed set of resources both with low cost and with good load balance.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design

Keywords

Load Balancing, Desktop Grid, Peer-to-Peer System

1. INTRODUCTION

The recent growth of the Internet and the CPU power of personal computers and workstations enables *desktop grid* computing to achieve tremendous computing power with low cost, through opportunistic sharing of resources [1, 2, 6]. However, traditional server-client grid architectures have inherent problems in robustness, reliability and scalability. Researchers have therefore recently turned to Peer-to-Peer (P2P) algorithms in an attempt to address these issues [5, 7, 9, 13].

Our goal is to design and build a highly scalable infrastructure for executing Grid applications on widely distributed sets of resources. Such infrastructure must be *decentralized*, *robust*, *highly available* and *scalable*, while effectively mapping application instances to available resources throughout the system (called *matchmaking*). By employing 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 or exceed the minimum job resource 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 [14], and a system such as BOINC [1] or SETI@HOME [2] for farming out jobs from a server to be run on a potentially very large collection of machines in a completely distributed environment.

However, efficiently matching heterogeneous jobs to heterogeneous computational resources becomes more challenging as such systems scale to large configurations and heavy workloads. Our previous work [12] addressed these issues and showed the tradeoffs between efficient matchmaking and good load balancing through a comparative analysis of three different matchmaking algorithms.

In this paper, we extend our previous work and describe algorithms and techniques that achieve both efficient matchmaking of jobs and good load balancing in decentralized and heterogeneous computational environments. The contributions of the paper are:

1. An intelligent matchmaking algorithm that is guaranteed to find a resource that meets the multiple requirements of a job, if such a resource exists somewhere in the system
2. Parsimonious resource usage that avoids wasting resources that are over-provisioned with respect to the jobs
3. Adapting the current load of the system to use more capable resources when the overall system is lightly loaded
4. Both efficient matchmaking and good load balancing with low cost

The rest of the paper is structured as follows. Section 2 discusses the context and overall goals of the work. Section 3 presents related work, while Section 4 describes the algorithms and optimization criteria for matching jobs to resources. Finally, Section 5 contains our evaluation, and Section 6 concludes.

[*]This research was supported by NASA under Grant #NNG06GE75G and the National Science Foundation under Grants #CNS-0509266 and #CNS-0615072.

2. ASSUMPTIONS AND GOALS

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

Our expected environment and usage make this problem easier in some ways and more difficult in others. A large fraction of nodes in the system might belong to one of a small number of equivalence classes in terms of their resource capabilities. 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. Node clusters make the problem more difficult by removing the notion of a single best match for a given job. The underlying matchmaking 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 matchmaking algorithm. Similarly, job profiles might show clustering in terms of their minimum resource requirements. Sets of similar jobs can result from running the same application code with slightly different parameters or input datasets. For example, researchers 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.

Therefore, the overall problem space for Grid computing environments can be divided along two axes, measuring the degree to which the nodes and jobs are either *clustered* or *mixed*. Systems such as Condor [14] mainly target mixed jobs in clustered nodes, while systems like BOINC [1] or SETI@Home [2] deal with clustered jobs in mixed nodes. Our intent is to effectively support all of these scenarios.

To summarize, the goals of any matchmaking algorithm must include the following:

1. *Capability* - The matchmaking framework should allow users to specify minimum requirements for any type of resource (CPU speed, memory, etc.).

2. *Load balance* - Load (jobs) must be distributed across the nodes capable of performing them.

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.

4. *Completeness* - A valid assignment of a job to a node must be found if such an assignment exists.

5. *Low overhead* - The matchmaking must not add significant overhead to the cost of executing a job. This may be challenging, given that the matchmaking is done in a completely decentralized fashion.

3. 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 most popular algorithms for object location and routing in P2P networks (called *Distributed Hash Tables* or DHTs [18, 19]) are capable of scaling to very large numbers of peers and simultaneous requests for service. A system can build upon these basic services to allow users to place idle computational resources into a general pool and draw upon the resources provided by others when needed.

Research such as [4, 9, 16] proposed a P2P architecture to locate and allocate resources in the Grid environment by employing a *Time-To-Live* (TTL) mechanism. TTL-based mechanisms are relatively simple but effective ways to find a resource (that meets the job requirements) in a widely distributed environment without incurring too much overhead in the search. However, such mechanisms may fail to find a resource capable of running a given job, even though such a resource exists somewhere in the network (lack of *Completeness*).

Studies on encoding static or dynamic information about computational resources using a DHT hash function for resource discovery have also been conducted [5, 8, 17]. However, there can be a load balancing problem for these encoding techniques, since a small fraction of the nodes can end up containing a large fraction of the resource capabilities of the nodes if there are many that have very similar (or identical) capabilities in the system (lack of *Load balance*). Also, simple encoding of resource information cannot effectively avoid selecting resources that are over-provisioned with respect to the jobs (lack of *Precision*).

The CCOF (Cluster Computing on the Fly) project [15, 21] 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 a Content-Addressable Network (CAN) [18] to use idle night-time cycles geographically distributed across the globe [22]. However, the host availability model in these work is not based on the resource requirements of the jobs (lack of *Capability*).

Awan et al. [3] 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 (which cannot achieve the efficiency of a DHT). By employing efficient uniform random sampling using random walks, probabilistic guarantees on the performance of the system could be achieved. However, as for the CCOF project, the job allocation model in this work does not consider the requirements of the jobs nor the varying resource capabilities of nodes in the system (lack of *Capability*).

4. MATCHMAKING ALGORITHMS

We begin by defining terminology and the basic framework of our approach to matchmaking, and then describe the details of the improvements we have made in our *CAN-based matchmaking framework*.

4.1 Overall System Architecture

All of the work described assumes an underlying distributed hash table (DHT) infrastructure [18, 19]. 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 using such an architecture, we effectively *reformulate* the problem of matchmaking to one of routing in the P2P network.

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 the system are *independent*, which implies that no communication is needed between them. This is a typical scenario

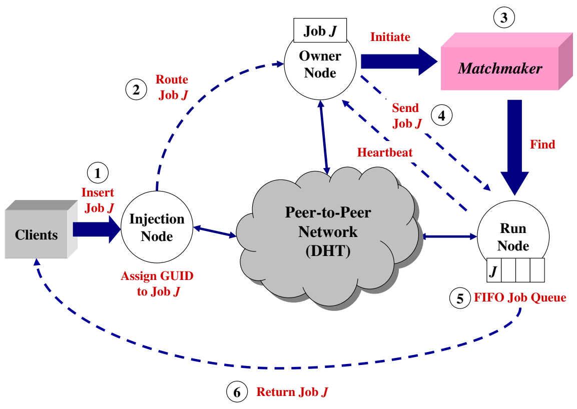


Figure 1: Overall System Architecture

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

Figure 1 shows the overall system architecture and flow of job insertion and execution in the P2P network. The steps of job execution are as follows:

1. A client inserts a job into a node in the system (*injection node*). The DHT provides an external mechanism that can find an existing node in the system [18, 19].

2. The injection node assigns a *Globally Unique IDentifier* (GUID) to the job by using its underlying hash function and routes the job to the *owner node*.

3. The owner node initiates a matchmaking mechanism to find a *run node* capable of running the job.

4. Once the matchmaking mechanism finds a run node for the job, the owner node sends the job to the run node.

5. The job is inserted into the job queue of the run node, which processes jobs in FIFO order. While processing the jobs, the run node periodically sends *heartbeat* messages to the owner node, which can relay the message to the client that initiated the job

6. When the job is finished, the run node returns the results to the client.

An owner node is responsible for monitoring the execution of the job and ensuring that its results are returned to the client. 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.

4.2 Basic Mechanisms

In this section, we briefly describe our basic approach to perform matchmaking based on a Content-Addressable Network (CAN) [18].

A CAN is a DHT that maps GUIDs of nodes and data to points in a d-dimensional space so that each node divides up the CAN space into rectangular *zones* and maintains *neighbor* information. The conventional use of CAN is to map a GUID into the space by applying d different hash functions, one for each dimension. However, positions in the CAN space need not be created through randomized hashes. For example, Tang et al. [20] map documents and queries into a CAN space where each dimension measures the relevance of a particular index term, executing queries via a blind local search centered on a query's mapping.

Similarly, we can formulate the 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 requirements on each resource type, respectively, to determine their coordinates. As a simple example, if the 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 just described. 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 matching node. Most importantly, job requirements represent *minimum* acceptable quantities. Any node meeting a job's requirements can run the job, but a node whose coordinate in any dimension is less than that specified by the job's requirements, even if very close in the CAN space, is not a viable choice to run the job. Hence our matchmaking/routing procedure must search for *the closest node whose coordinates in all dimensions meet or exceed the job's requirements*.

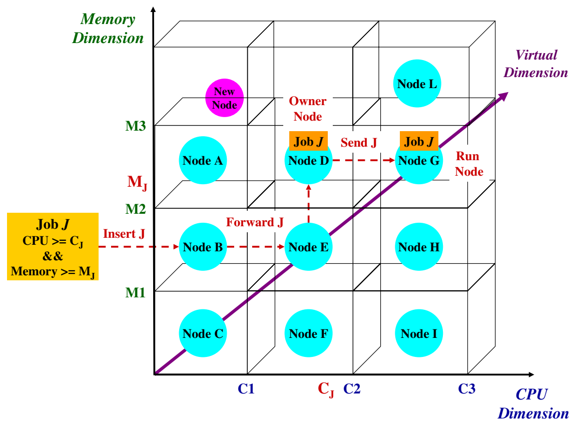


Figure 2: Matchmaking Mechanism in Basic CAN

Figure 2 shows the procedure for matching a job J to the Node G in a system with two resource types, CPU speed and Memory size, through routing in the CAN space. A job is inserted into the system using its requirements as coordinates ($\{C_J, M_J\}$ for Job J) and defining the owner of the resulting zone as the owner node of the job (Node D). The owner node creates a list of candidate run nodes, and chooses the (approximately) least loaded among them (Node G) based on load information periodically exchanged between neighboring nodes. The candidate nodes are drawn from the owners of neighboring zones, such that each candidate is at least as capable as the original owner node in all dimensions (capabilities), but more capable in at least one dimension (Nodes G and L).

The above procedure works in all cases, but may cause some problems for the CAN mechanisms 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 (New Node and Node A in the figure). The best way to distribute ownership of a zone across multiple such nodes is not immediately obvious. Conversely, many jobs might have very similar requirements. For example, many jobs will likely be inserted into the system with no requirements at all specified. In this case, all of those jobs will be mapped to a single node that owns the zone containing the minimum point in the CAN volume (Node C in the figure).

We address this problem 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 a randomly generated virtual dimension value. Therefore, even when multiple identical nodes join the system, they are mapped to distinct locations, and CAN 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 requirements and a randomly assigned virtual dimension coordinate. In combination, the randomly assigned node and job coordinates act to break up clusters and spread load more evenly over nodes. More details can be found in our previous work [12].

4.3 Improvements

In previous work, we showed that the CAN-based matchmaking mechanism can achieve good load balancing among the multiple candidate run nodes with low matchmaking cost in most scenarios. However, we found that in certain circumstances the CAN-based algorithm works very poorly due to serious load imbalance when jobs with few requirements are run on nodes with heterogeneous (mixed) resource capabilities. For example, suppose we have a hypothetical CAN with only a single real dimension, CPU speed. If most jobs do not specify CPU requirements, their CPU speed coordinates will have the minimum value in that dimension. The jobs can still be mostly distributed (via the virtual dimension) along a line at a single CPU coordinate. However if most nodes have distinct CPU speeds (mixed node profiles), the slowest node ends up covering the bulk of the virtual dimension at low CPU speed, and will become the owner of a disproportionate number of the jobs, resulting in load imbalance [12].

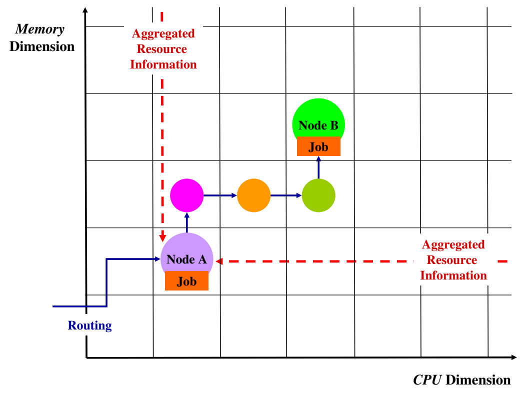

Memory Dimension
Aggregated Resource Information
Node B
Job
Node A
Job
Aggregated Resource Information
Routing
CPU Dimension

Figure 3: Improving the CAN-based Mechanism

We now describe how we have improved the basic CAN-based matchmaking mechanism to address this problem by *pushing* jobs into underloaded regions of the CAN space based on *dynamic aggregated load information*.

Figure 3 shows the basic concepts of our improvements. When a new job is inserted into the system and routed to the owner node (Node A), the job is *pushed* into an underloaded region in the CAN space. To determine whether to initiate pushing of a job, a fixed amount of current system load information is propagated along each dimension in the CAN space. If the overall system is lightly loaded, the job can be pushed into the upper regions of the CAN space (farther from the origin) and utilize the more capable nodes in the system (Node B). We cannot push jobs to lower regions (closer to the origin) in the CAN space, because the nodes occupying those regions will likely not be able to satisfy the jobs' requirements. It is very important that each node in the pushing path of a job be able to make the decision whether to continue pushing the job in a completely decentralized fashion, based only on local information. Therefore, the amount of information maintained by each node for pushing jobs should remain *constant* with respect to the number of jobs.

4.4 Enhanced CAN Mechanism Details

To enable the pushing of a job to an underloaded region in the CAN, we have to propagate a fixed amount of current load information through the nodes in the CAN space. Since each node cannot maintain an accurate global picture of the system load, the load information must be properly *aggregated*. Also, the load information should be *dynamic* so that it can reflect the current distributed state of the system. For this dynamic aggregated load information we use the following measures along *each* dimension in a CAN space:

- *Number of Nodes*
- *Sum of the Job Queue Sizes*

We add this aggregated load information to the *periodical neighbor state update* mechanism of the original CAN DHT maintenance algorithm [18], to avoid generating additional messages in the P2P network. By using the two aggregated load statistics, for a given node N we can estimate the current load (e.g., average job queue size) along each dimension of the CAN for the nodes that own CAN regions with greater values than that of node N in that dimension. However, it is not easy to accurately compute the aggregated load information, since the overall CAN space can be *irregularly* partitioned. To build a regularly partitioned CAN space, the representative points for all nodes in the system should be distributed uniformly. In our CAN, the point for a node consists of its resource capabilities and an additional virtual dimension coordinate. Therefore we cannot assume that the resource capabilities of the nodes in the system have a uniform distribution since in the real system, only a small portion of the nodes are likely to have high resource capabilities, with the majority of the nodes having relatively lower capabilities [22].

To deal with aggregation of load information in the irregular CAN space, the algorithm uses an *overlap fraction*-based computation, as shown in Figure 4. Figure 4 shows the process for aggregating load information along the Memory dimension in a CAN space. AggrInfo(N) is the computed aggregated load information from nodes with Memory values greater than that of node N (`Number of Nodes` or `Sum of the Job Queue Sizes`). Info(N) is the current load information for node N (e.g., job queue size of N). Whenever a node N computes its aggregated load information, it only carries some *fraction* of the information from its neighbors

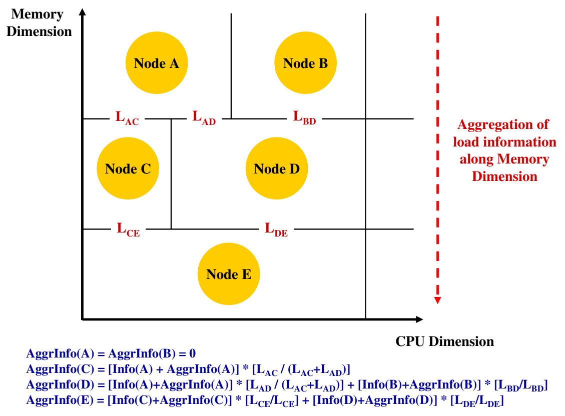

Memory Dimension
Node A
Node B
L_{AC}
L_{AD}
L_{BD}
Node C
Node D
Aggregation of load information along Memory Dimension
L_{CE}
L_{DE}
Node E
CPU Dimension
AggrInfo(A) = AggrInfo(B) = 0
AggrInfo(C) = [Info(A) + AggrInfo(A)] * [L_{AC} / (L_{AC}+L_{AD})]
AggrInfo(D) = [Info(A)+AggrInfo(A)] * [L_{AD} / (L_{AC}+L_{AD})] + [Info(B)+AggrInfo(B)] * [L_{BD}/L_{BD}]
AggrInfo(E) = [Info(C)+AggrInfo(C)] * [L_{CE}/L_{CE}] + [Info(D)+AggrInfo(D)] * [L_{DE}/L_{DE}]

Figure 4: Computing Aggregated Load Information

with larger Memory values, depending on how much *N*'s boundary overlaps with those neighbors. Note that the information about the neighbors is propagated through the periodical CAN neighbor state update mechanism. More generally, for each dimension *d* in a CAN space, node *N* can compute the aggregated load information along the dimension *d* (denoted by $AI_d(N)$) as follows:

$$AI_d(N) = \sum_{u \in UN_d} (AI_d(u) + I(u)) \times OF_d(N,u) \tag{1}$$

$$OF_d(N,u) = \frac{\prod_{i \neq d} OverlapEdge(u,N,i)}{\prod_{i \neq d} Edge(u,i)} \tag{2}$$

In Equation 1, UN_d is the set of nodes adjacent to N with which it shares a border along N's upper edge in dimension d. For Node D in Figure 4, and considering the Memory dimension, this would be the set $\{$Node A, Node B$\}$. For each node u in UN_d, N adds the local and aggregated information from u and multiplies it by a factor $OF_d(N,u)$. This factor reflects the fact that nodes other than N might have u as a neighbor in dimension d (for example, Node C also has Node A as a neighbor), so without the multiplier u's information will be included more than once (when Node E aggregates information from both Node C and Node D). In particular, if LN_d are the lower neighbors of u at dimension d (thus $N \in LN_d$), then it must hold that

$$\sum_{v \in LN_d} OF_d(v,u) = 1$$

in order for u's load information to be aggregated in full along dimension d (Node A's information must be split between Node C and Node D).

The aggregation multiplier $OF_d(N,u)$ is the *overlap fraction* of N and u along dimension d, from the perspective of node u. That is, if N and u control adjacent *hyper-volumes* in the CAN space, it is the fraction of u's *hyper-area* at its lower bound in dimension d that intersects with N's hyper-area at its upper bound in d. In two dimensions, it is the length of the line segment describing N and u's shared border divided by the full length of u's bordering edge. For example, $OF_{Memory}(D,A) = L_{AD}/(L_{AC}+L_{AD})$, where *L* is the length of the line segment. In higher dimensions, the orthogonality of the dimensions means that we can compute each of these linear fractions for the dimensions other than d, and take their product to obtain the overlap fraction. This is what is shown in Equation 2, where $OverlapEdge(u,N,i)$ is the overlap of u and N in dimension i (L_{AD} for Node D and Node A in the CPU dimension) and $Edge(u,i)$ is the length of u's edge in dimension i ($L_{AC}+L_{AD}$ for Node A in the CPU dimension).

Once the aggregated load information is propagated through the entire CAN space, all the way to the nodes near the origin, the system is able to push the incoming jobs into underloaded regions for better load balancing and to utilize more capable nodes in the system. To initiate the job pushing we have to address several issues as follows:

1. *Target Node* - Where should a job be sent?
2. *Stopping Criteria* - When should pushing be stopped?
3. *Criteria for the Best Run Node* - Which candidate run node should be selected?

To determine the *target node*, first we want to push the jobs into lightly loaded regions of the CAN space. Likely the best way to determine the load of the system is to use the *aggregated average job queue size*. Since each node has aggregated load information about each upper neighbor locally, it can calculate the aggregated average job queue size for each upper neighbor by using `Number of Nodes` and `Sum of the Job Queue Sizes` carried by the load propagation mechanism. However, the shortest average job queue size does not always give the best choice. A node with a slightly longer aggregated average queue size might also enable access to a larger number of potential run nodes than the node with the smallest aggregated average queue size. This larger number of nodes makes it more likely that when a pushed job reaches one of the nodes believed to be lightly loaded, that node will still be lightly loaded. Therefore, we want to push jobs to the upper neighbor node that has both a small aggregated load (average job queue size) and a large number of available nodes above that neighbor node, to increase the number of candidate run nodes. To summarize, we can determine the target node based on the following objective function:

$$F_d(u) = \frac{AI_d(u).SumOfJobQueueSizes}{(AI_d(u).NumberOfNodes)^2} \tag{3}$$

Whenever a node chooses a target node from among its upper neighbors, it calculates $F_d(u)$ for each $u \in UN_d$ and picks the one that has the minimum objective function value across all dimensions.

By using the objective function in Equation 3, each node in the path of a pushed job can decide where to push the job based only on local information. The question then is the *stopping criteria* – *when* should pushing be stopped? We must avoid pushing jobs to the extreme edges of the CAN space, because that will result in load imbalance. The stopping criteria for pushing a job should reflect the current (but distributed) load of the system and be computed based only on each node's local information. The very first condition for stopping should be whenever the matchmaking mechanism finds a *free* node that meets the resource requirements of a job; then matchmaking can stop pushing the job and assign the free node as the run node. Note that each node can determine whether there is a free node in its neighborhood based only on its local neighbor state information, which is updated periodically. In a relatively lightly loaded system, this mechanism works well, since every time the matchmaking is performed, it can find a free node in the system. However, in a heavily loaded system where most, if not all, of the nodes are already busy processing jobs, it is not clear how we stop pushing a job without causing severe load imbalance. A simple way to do this is for each node to estimate the current load (average job queue size) of its surrounding neighbors, and if the load is below a predefined *threshold*, then it can stop pushing and assign the job to one of its neighbor nodes. However, to determine a threshold that is insensitive to the characteristics of various workloads is not trivial. Therefore, we employ *probabilistic stopping* according to the following formula:

$$PS(N) = \frac{1}{(1 + AI_{TD}(N).NumberOfNodes)^{SF}} \qquad (4)$$

In Equation 4, $PS(N)$ shows the probability to stop pushing a job from node *N*, and SF is the *stopping factor*, which greatly affects the shape of the probability function. As the number of nodes above node *N* in the target dimension TD (determined by the neighbor minimizing Equation 3) becomes smaller, the probability of stopping becomes greater. This means that if a job approaches the edges of the CAN space, with high probability the pushing will stop and a run node chosen based on local information. This feature avoids pushing incoming jobs to the edges of the CAN space, which would overload the nodes near the edges. We can adjust the probability function by changing SF (higher SF means a higher probability of pushing the job). We tested three different SF values from 1 to 3 and show the experimental results in Section 5.

We have shown (1) how to aggregate the dynamic load information in a CAN space (Equations 1 and 2), (2) based on that information how to choose a target node for a job (Equation 3), and (3) when to stop pushing a job (Equation 4). The final step in the matchmaking algorithm is to choose the *best* run node among the multiple candidates. Pushing of incoming jobs can be stopped either because the matchmaking mechanism found a free node or due to the probabilistic stopping function. In the former case, the node where the pushing stopped (we call this node the *matching node*) creates a list of capable candidates using its local neighbor state information. It is possible that there might be multiple free nodes among the candidates, in which case the matchmaking algorithm selects the *fastest* candidate run node (measuring CPU speed), since that can speed up the overall processing of a job. However, if the pushing process stopped because of the probabilistic stopping function, this means that there are not enough free nodes in the system. To choose the best run node from among the candidates, but with no available free nodes, we use the following score function for ranking the candidates:

$$F(C) = \frac{C.JobQueueSize}{C.SpeedOfCPU} \qquad (5)$$

In Equation 5, $F(C)$ is the score function for a candidate run node C. The candidate node with the minimum score will be selected as the best run node: the algorithm prefers a node with a smaller job queue and a faster CPU. Using only the set of candidate run nodes built by the matching node may not be sufficient, since we are pushing the jobs across multiple nodes in the system. Therefore, we still consider the candidate run nodes found in the process of pushing, in addition to the candidate run nodes around the matching node, for better load balancing. To summarize, at each step of pushing a job, the matchmaking mechanism keeps the best candidate run node based on the score function in Equation 5, and considers it in the list of candidates created by the matching node whenever the matchmaking mechanism cannot find a free node in the system.

5. EVALUATION

In this section, we evaluate our matchmaking algorithms in decentralized and heterogeneous environments through a comparative analysis of experimental results obtained via simulations. To compare against our CAN-based approach, we evaluate two additional matchmaking algorithms, a *Rendezvous Node Tree*-based approach and a *Centralized Matchmaker*, that were described in detail in our previous work [10, 12].

5.1 The Rendezvous Node Tree

We briefly introduce the Rendezvous Node Tree (RNT), which uses a distributed data structure built on top of an underlying Chord DHT [19]. An RNT contains all participating nodes in the desktop grid. Each node determines its parent node based only on local information, which enables building the tree in a completely decentralized manner. Due to the uniform distribution of GUIDs of the nodes in the system, the overall height of the RNT is likely to be $O(\log N)$ where *N* is the total number of live nodes in the system (see details in Kim et al. [10]). Once the parent-child relationship in the RNT 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 RNT (*hierarchical aggregation*)

We inject jobs into the system by mapping each to a randomly-chosen node that becomes the job's owner node, which achieves a good initial load balancing by spreading the jobs across the system. The owner node initiates a search for a node on which to run the job. The search first proceeds through the subtree rooted at the owner node, only searching up the tree into subtrees rooted at the ancestors of the owner node if the subtree does not contain any satisfactory candidates. The search is *pruned* using the *maximal amount of each resource available* up and down the tree carried by the hierarchical aggregation mechanism. Rather than stopping at the first candidate capable of executing a given job, the search proceeds until at least k capable nodes are found for better load balancing (*extended search*). If any of the capable nodes has an empty queue, the empty node with the fastest CPU is selected. Otherwise, the candidate node chosen is the one with the smallest value of the score function shown in Equation 5.

The RNT-based approach has a different underlying rationale than that of the CAN-based mechanisms [12]. Specifically, the RNT copes with dynamic load balance issues by performing a tree traversal after the initial mapping, and addresses *Completeness* by passing information describing the maximal amount of each resource available up and down the tree (*matchmaking after load balancing*). Therefore, RNT is a good comparison model for our CAN-based matchmaking frameworks which employ load balancing techniques after approximate mapping of jobs to the nodes in the system (*load balancing after matchmaking*).

5.2 Centralized Matchmaker

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, and so can assign a job to the node that both satisfies the job requirements and has the lightest current load across all nodes in the entire system. 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 is a good comparison target for other matchmaking algorithms (as in Oppenheimer et al. [17] and Zhou et al. [21]).

We can view the Centralized Matchmaker algorithm as the extreme case of the RNT or CAN based search algorithm, since it first finds *all* candidate run nodes that meet the job requirements and picks the one with the lightest load. However, such a scheme would not be feasible in a completely decentralized system implementation, since the algorithm would incur a large overhead to find *all* nodes in the P2P system that meet the job requirements, and the node performing the centralized algorithm would be a single point of failure for the system.

5.3 Experimental Setup

We use synthetic job and node mixes to simulate the behavior and measure the performance of our improved CAN-based matchmaking algorithm. 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, node departures (graceful or from a failure), and job submissions. The events are generated using a Poisson distribution with an arrival rate of $1/\tau$ (τ is the average event inter-arrival time). 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 Zhou et al. [22]).

Our test traffic workloads differ on two axes. Workloads are categorized as either *clustered* or *mixed* (as described in Section 2). 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 are "lightly" or "heavily" constrained. For a given job, each type of resource has a fixed independent probability of being constrained: "lightly-constrained" jobs have an average of 1.3 constraints (out of the 3) and "heavily-constrained" jobs have an average of 2.4. As a job has more minimum resource requirements (heavily-constrained workloads), it is likely to be harder to match the job since fewer nodes in the system can meet those multiple constraints.

In this paper, we only present results from *mixed* workloads since in the clustered workloads, the CAN-based matchmaking mechanism already has shown better performance than the RNT-based approach and is close to that of the Centralized Matchmaker [12].

The amount of work *W* for a job *j* is generated uniformly at random from a predefined set of work ranges (40 minutes on average), and means that to run the job *j* a node must execute for *W* time units if it has *exactly* the same node specification as does the job *j*'s constraints. To model the actual running time of a job, we divide *W* by the node CPU speed (relative to some baseline node CPU speed), to get a run time on the node a job is assigned to. Finally, for the network communication cost, the latency of a packet between any two nodes in the system is modeled by an exponential distribution with a mean of 50 milliseconds.

Our metrics are *matchmaking cost* (the amount of time between when a job is injected and when it is assigned to a run node in the system), *wait time* (the amount of time between when a job is injected and when it actually starts running) and *average queue length* (the length of the non-preemptive job queue seen by a job when it is finally assigned to a run node). Matchmaking cost directly quantifies the overhead 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 job queue of a run node before a job is executed. Wait time reflects both protocol overhead and the quality of the matchmaking results, i.e., load balancing. Finally, the distribution of queue lengths provides a direct measurement of the load balance seen by injected jobs.

We test the original CAN approach (Section 4.2) (**CAN**) and the improved CAN approach employing dynamic aggregated load information (Section 4.3 and 4.4) with different stopping factors from 1 to 3 (**CAN-P1,2,3**). To compare against CAN-based matchmaking mechanisms, we also tested the RNT-based approach (Section 5.1) (**RNT**) and the idealized centralized approach (Section 5.2) (**CENTRAL**). We do not include matchmaking cost for the centralized approach because it incurs no cost for matchmaking.

5.4 Performance Results

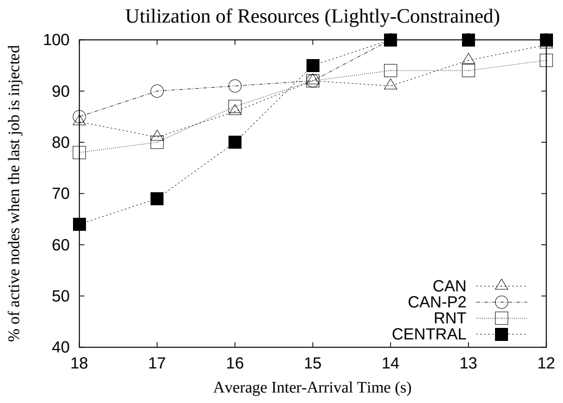
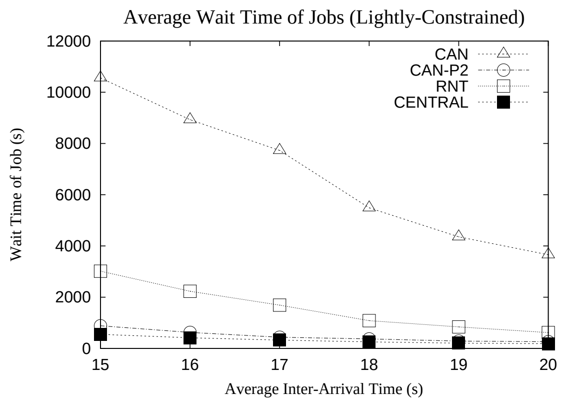


Figure 5: Utilization of Resources for Lightly-Constrained Workloads

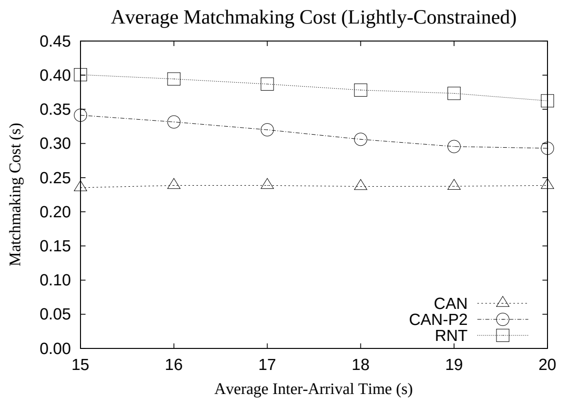
We begin by discussing the experimental results obtained from relatively static workloads with lightly and heavily-constrained jobs, respectively. In the static workloads, no nodes join or leave the system during the course of the experiments. There are six different workloads for the lightly-constrained jobs, which have different values of τ from 15 seconds to 20 seconds. Similarly, for the heavily-constrained workloads, we varied τ from 25 seconds to 30 seconds.

The important characteristic of these workloads is that all of them reach a *steady state* during the simulation period. For example, the percentage of *active* nodes (nodes currently running jobs) when the last job is injected into the system for lightly-constrained workloads is depicted in Figure 5. Figure 5 shows that for values of τ from 18 down to 16 seconds, the utilization of the overall system resources remains low, indicating lightly loaded environments, while from 14 seconds down almost 100% of the nodes are busy processing other jobs when the last job is inserted into the system. This means the system has reached its maximum throughput. Interestingly, the utilization of CENTRAL is smaller than all other matchmaking mechanisms in lightly loaded environments (from 18 to 16 seconds). This is because CENTRAL is the global algorithm that can assign a job to the fastest idle node in the system, which accelerates the rate at which jobs are processed.

In the steady state, the rate for incoming jobs and finishing jobs is approximately the same, and we want to show the performance of each matchmaking mechanism in this steady state, to avoid the transient effects of earlier jobs that see a largely empty system. We can inject more jobs with smaller τ to increase the system load, which will eventually saturate the system and result in indefinite



(a) Average Job Wait Time



(b) Average Matchmaking Cost

Figure 6: Experimental Results for Lightly-Constrained Workloads

growth of job queues. However, this will not be feasible in a real system, since when the overall system becomes too heavily loaded the system can *refuse* to receive more jobs until it becomes stabilized. The desire to measure steady state behavior explains why we choose different ranges for τ for lightly and heavily-constrained jobs. In the heavily-constrained workloads, many jobs have multiple resource requirements, and this reduces the number of nodes that are legal matches for a job in the system. Therefore to make the workloads reach steady states, we increase τ for these jobs relative to the lightly-constrained workloads. The workloads belonging to either the lightly or heavily-constrained sets have *exactly the same* job and node profiles, respectively, so that we can directly compare across different values of τ.

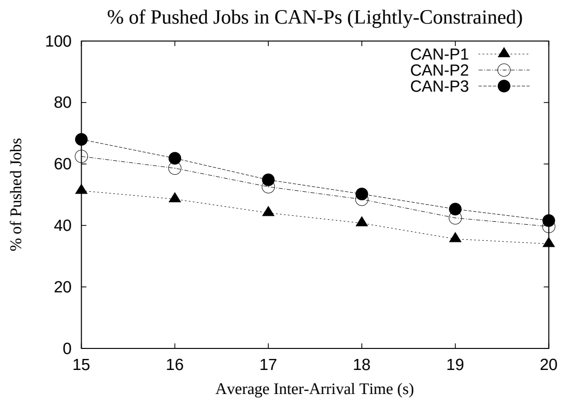
In this paper, we only present results from *lightly-constrained* workloads since for the heavily-constrained ones we verified that most of the behaviors and performance of the CAN-based matchmaking algorithms are similar to those in our previous work [12]. We refer interested readers to our extended version of this paper [11].

Figure 6(a) shows the performance results for the matchmaking mechanisms, measuring job wait time for lightly-constrained workloads. We omitted the results for average queue lengths since they show similar behavior to the job wait time metric [11]. This is because a majority of the job wait time consists of waiting time in the job queues, which shows the importance of load balancing. We only plot the improved CAN-based matchmaking mechanism with stopping factor 2 (CAN-P2) since it shows relatively stable performance for both lightly and heavily-constrained workloads (insensitive to the characteristics of the workloads). The results imply that our improved CAN-based matchmaking mechanism shows very competitive performance even compared to CENTRAL and improves the quality of load balancing dramatically from the original CAN algorithm (CAN). More specifically, CAN-P1 has 2.1 times the average job wait time of CENTRAL across all the lightly-constrained workloads, CAN-P2 is a factor of 1.5 worse and CAN-P3 is a factor of 1.4 worse, while the RNT is a factor of 4.6 worse and CAN is 21.2 times worse. The main reason CAN has poor load balancing is that for the lightly-constrained workloads, a majority of the jobs has few or no constraints, so that many jobs are mapped to a comparatively small region of the CAN space near the origin. More specifically, if a job does not specify any requirement for a specific resource type, the corresponding coordinate for the job is mapped to the minimum constraint value (in our case, 0), and this results in a *hot spot* causing load imbalance. However, by pushing jobs to underloaded regions of the CAN space, CAN-P2 can disperse the jobs in the different dimensions from the original hot spot, which results in superior load balancing (as seen in Figure 6(a)). Additionally, CAN-P2 can utilize more capable nodes whenever needed, which can accelerate overall job processing so that CAN-P2 also outperforms the RNT.

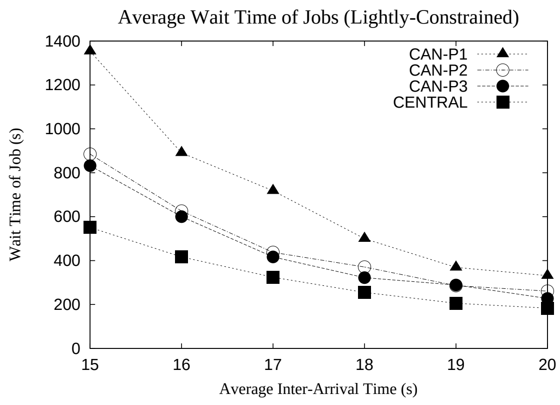
However, pushing jobs in the CAN space may cause additional overhead for matchmaking, since each job must traverse the CAN space from its owner node to find an appropriate run node. Figure 6(b) shows that CAN-P2 has worse matchmaking performance than CAN. Also, as we increase the stopping factor (SF), the matchmaking cost increases accordingly, since with higher SF the probability for stopping decreases. However, all of the CAN-based matchmaking mechanisms (CAN and CAN-P2) still show better matchmaking performance than RNT. This is because the CAN-based matchmaking mechanism inserts each job into the right place in the DHT for matchmaking (the owner node), where surrounding neighbor nodes can already meet the resource requirements of the job. However, in the RNT approach each job starts from a completely random place in the DHT and must find an appropriate run node for the job through searching up and down the RNT. Another interesting result in Figure 6(b) is that all of our matchmaking algorithms (including CAN, CAN-P2 and RNT) show very low cost for performing matchmaking in distributed and heterogeneous environments. Compared to the wait time of jobs shown in Figure 6(a), the cost for matchmaking is negligible. This could be because of our assumption about the average packet delay for a message, which is set to 50 milliseconds. However, ignoring the packet delay, the results show that all of our matchmaking mechanisms find an appropriate run node with a very small number of P2P network hops to achieve good load balancing. Hence, we can concentrate on the load balancing issue whenever the average running time of jobs (in our case, 40 minutes) is significantly longer than the network communication speed, which is a typical scenario in a desktop grid computing environment.

Costs and Benefits of SF.

Different stopping factor values can affect the behavior of the CAN-P algorithm, as measured by the number of jobs pushed, as seen in Figure 7(a). With higher SF, more jobs will be pushed into the upper regions of the CAN space due to the decreased stopping probability, so that CAN-P3 shows the highest percentage of pushed jobs among the three different CAN-Ps. Increasing the



(a) % of Pushed Jobs



(b) Average Job Wait Time

Figure 7: Costs and Benefits of CAN-P for Lightly-Constrained Workloads

stopping factor increases the overall matchmaking cost, since jobs are pushed farther in the CAN space to find appropriate run nodes. However, that does provide benefits from better load balancing, as seen in Figure 7(b), since more capable nodes end up being used for some jobs in the system. As the overall system becomes lightly loaded (increasing τ), the percentage of pushed jobs decreases, since the matchmaking mechanism is more likely to encounter an empty node (as seen from Figure 7(a)). The decrease is less for heavily-constrained workloads since there are not as many nodes in the system that can run the incoming jobs, which means that the jobs start pushing from relatively near the edges of the CAN space.

Dynamic Workloads.

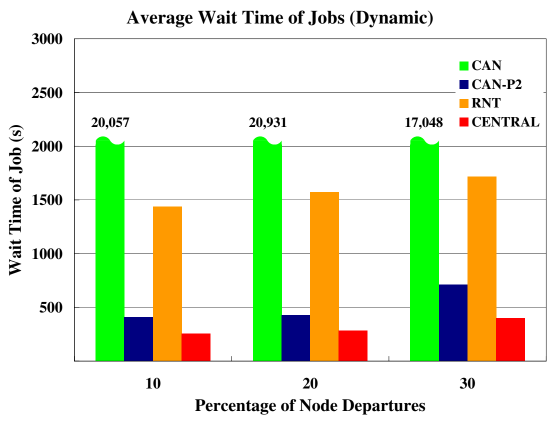


Figure 8: Experimental Results for Lightly-Constrained Dynamic Workloads

Figure 8 shows wait times for three *lightly-constrained mixed* workloads, where between 10% and 30% of the nodes leave during the course of simulation, and shows that node departures can affect CAN-P's ability to match CENTRAL's performance. The value of τ for all of the dynamic workloads is set at 17.5 seconds. Note that in Figure 8, results from the basic CAN are truncated since they have very large values compared to the other matchmaking frameworks. Node departures include graceful departures, where a node informs its neighbors before leaving, and failures, where the neighbors learn of the departure from missing P2P network heartbeat messages. All of the dynamic workloads have the same number of jobs and the same job profiles, but have different sets of available nodes in the system at different times, so that we cannot directly compare across workloads.

In the dynamic workloads, because existing nodes depart the system the information carried by the CAN- and RNT-based mechanisms can be more stale compared to the information maintained for static workloads, and there can also be some overhead for P2P network recovery (unlike for CENTRAL). More specifically, CAN-P2 shows 1.6 times the job wait time of CENTRAL on average across all the workloads, and RNT is a factor of 5.2 worse. Although we cannot directly compare these results with Figure 6, clearly there are some load balancing issues for both the CAN-P and RNT algorithms, that keep them from approaching the wait time performance of CENTRAL. The dynamic behavior of the nodes in the system seems to have a much larger impact on basic CAN compared to CAN-P2 or RNT. Since all of the dynamic workloads are based on mixed sets of nodes and jobs, a load imbalance problem similar to the one that we saw for the basic CAN earlier, due to a hot spot in the CAN space, can occur as the jobs are entering the system and being assigned to run nodes. However if one of the nodes in the hot spot leaves the system or fails, that can be disastrous for wait time performance, since all of the jobs that were running or waiting in the departed node must be re-assigned to other live nodes in the system. Since each node in the hot spot has a disproportionate number of assigned jobs, this causes even more severe load imbalances. However, by employing the pushing mechanism based on dynamic aggregated load information, CAN-P2 can spread the jobs away from the hot spot and achieve more reliable load balancing compared to CAN and still outperforms the RNT, which is based on random initial load balancing.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a matchmaking framework for desktop grid systems that can effectively match incoming jobs and balance the load across multiple candidate nodes, without centralizing information or control. By extending our previous work [12], we have improved the CAN-based matchmaking mechanism to employ dynamic aggregated load information and to push jobs to underloaded regions of the CAN space. Through a comparative analysis of the experimental results obtained via simulations, we have shown that our system can reliably execute Grid applications on a widely distributed set of resources with good load balancing and low matchmaking cost.

Our work up to now has mainly considered *continuous* constraints for a job, such as minimum required CPU speed and memory size. However, we must also deal with *discrete* constraints for a job, such as operating system type and version. These kinds of discrete constraints can make the matchmaking process more difficult, since we have to find both exact matches for discrete constraints, and approximate matches for continuous constraints in a single protocol. Addressing this problem is a subject of future work. We are also in the process of building a prototype system based on CAN-P matchmaking, and will characterize its behavior on real workloads, via consultation with our application-area collaborators in physics and astronomy. In the future, we will measure and report on the behavior of our system for heterogeneous environments running real applications.

7. REFERENCES

[1] D. P. Anderson, C. Christensen, and B. Allen. Designing a Runtime System for Volunteer Computing. In *Proceedings of the 2006 IEEE/ACM SC06 Conference*, Nov. 2006.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, Nov. 2002.

[3] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Unstructured Peer-to-Peer Networks for Sharing Processor Cycles. *Parallel Computing*, 32(2), Feb. 2006.

[4] 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.

[5] 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.

[6] 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*, 63(5):597–610, May 2003.

[7] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[8] R. Gupta, V. Sekhri, and A. K. Somani. CompuP2P: An Architecture for Internet Computing using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306–1320, Nov. 2006.

[9] A. Iamnitchi and I. Foster. A Peer-to-Peer Approach to Resource Location in Grid Environments. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, pages 413–429. Kluwer Academic Publishers, 2004.

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

[11] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids (Extended Version). Technical Report CS-TR-4863 and UMIACS-TR-2007-16, University of Maryland, Department of Computer Science and UMIACS, Mar. 2007.

[12] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Resource Discovery Techniques in Distributed Desktop Grid Environments. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID 2006)*, Sept. 2006.

[13] 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.

[14] 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.

[15] V. Lo, D. Zhou, D. Zappala, Y. Lin, and S. Zhao. Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, Feb. 2004.

[16] C. Mastroianni, D. Talia, and O. Verta. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. In *Proceedings of the European Grid Conference (EGC2005)*, Feb. 2005.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] D. Zhou and V. Lo. WaveGrid: a Scalable Fast-turnaround Heterogeneous Peer-based Desktop Grid System. In *Proceedings of the 20th International Parallel & Distributed Processing Symposium*, Apr. 2006.