Integrating Categorical Resource Types into a P2P Desktop Grid System [*]

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

Abstract

We describe and evaluate a set of protocols that implement a distributed, decentralized desktop grid. Incoming jobs are matched with system nodes through proximity in an N-dimensional resource space. This work improves on prior work by (1) efficiently accommodating node and job characterizations that include both continuous and categorical resource types, and (2) scaling gracefully to large system sizes even with highly non-uniform distributions of job and node types. We use extensive simulation results to show that the resulting system handles both continuous and categorical constraints efficiently, and that the new scalability techniques are effective.

1 Introduction

Desktop grid computing systems have achieved massive computing power with low cost by leveraging unused capacity on high-performance personal computers and workstations across the Internet [1]. However, traditional centralized 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 [4, 11]. Our previous work [6, 7] described a matchmaking approach based on proximity between job and node characterizations in an *N*-dimensional Content-Addressable Network (CAN) [9], where each resource type corresponds to a distinct dimension. However, this approach only accommodates *continuous* resource characterizations, such as memory or disk size, or CPU speed. For continuous resource types, matchmaking requires that a node meet or exceed a job's requirements.

This work has two novel contributions. First, we extend our prior work to include *categorical* resource constraints. Categorical constraints require a singular value for that resource, such as a specific type of operating system or processor, as opposed to the minimum requirements for a continuous resource constraint. While such resource types could be accommodated by performing an additional check to match the categorical constraints (after identifying the nodes that are able to run the job since they meet the continuous resource requirements, as described in our previous work), that approach can be inefficient. For example, when the required categorical resources are not common, or not spread uniformly through the CAN space, a large number of such checks might have to be performed for each job. Another approach would be to create distinct CAN spaces for each combination of categorical constraints. However, the number of such spaces could be very large (the product of the numbers of different values for each categorical constraint), and the resulting system would be very inefficient for matchmaking in common cases, such as when not all categorical resource types are specified for a job (a "don't care" condition). Our approach is to integrate the categorical resource types into a single N-dimensional CAN space so that matchmaking is efficient for *any* combination of categorical and continuous constraints.

The second contribution of this paper is a set of optimizations to the basic CAN infrastructure that allows the system to scale well for highly non-uniform distributions of jobs and nodes. The load on individual nodes in a desktop grid consists of application load (the jobs to be executed), and system load (load imposed by the workings of the underlying system). Unfortunately, non-uniform distributions can cause any CAN, not just the modified CAN of our system, to distribute the system load unevenly across nodes. For example, consider a system with only two types of nodes, but many of each. The basic CAN infrastructure can cause a single node of one type to be a "neighbor" of all nodes of the second type, causing the system load incurred at that one node to scale linearly with the size of the system. Our optimizations distribute this load more fairly, without impacting the overall reliability or performance of the system.

2 Background

A general-purpose desktop grid system must accommodate various scenarios of node capabilities and job requirements. Nodes may be added one at a time over time, so that

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

their resource capabilities are heterogeneously distributed, or they may be added as sets of homogeneous clusters. Likewise, jobs may be relatively unique in their requirements, or part of a series of requests with similar or identical requirements (e.g., a simulation sweeping over a large set of parameter combinations). A good matchmaking algorithm must be expressive enough to fully describe both minimum job requirements and disparate nodes. Further, such an algorithm should evenly balance load across system nodes, and find a valid assignment for every job, if such an assignment exists. Also, 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. Finally, the matchmaking process should not add significant overhead to the cost of executing a job.

2.1 Overall System Architecture

In previous work [6, 7] we have found that a Content-Addressable Network (CAN) [9] provides a good framework for a decentralized desktop grid. A CAN is a type of distributed hash table (DHT) that maps nodes and jobs into a multidimensional space. In our case, nodes are mapped by their resource capabilities (each resource type is a separate dimension), and jobs by their resource requirements. The semantics of routing in a CAN places a job at a node that is minimally capable of running that job. The task of choosing a node to run the job proceeds from that point. All jobs in the system are *independent*, which implies that no communication is needed between them.

The steps involved in executing a job are as follows:

1. A client inserts a job into the system via some (arbitrary) node, called the *injection node*.
2. The injection node initiates CAN routing of the job, which ultimately places it at the job's *owner node*.
3. The owner node begins the matchmaking process, in which it looks for a lightly loaded node satisfying all of the job's requirements. This is the *run node*.
4. The run node places the job into a FIFO queue for eventual processing. Periodic soft-state heartbeat messages between the run and owner nodes ensure that both are still alive. Failure of either node prompts the other to select a replacement.
5. Once the job finishes, the run node returns the results to the client and informs the owner node (to terminate the heartbeats).

As nodes are mapped into the CAN space, each is assigned a non-overlapping hyper-rectangular *zone*. Each node maintains a list of *neighbors*, defined as those nodes whose zones abut its own. CAN routing is a greedy algorithm, in which a node passes a message (containing, for example, a job profile) to the adjoining zone that minimizes the distance to the message destination.

We augmented the basic matchmaking approach in two ways. First, the basic CAN procedure encounters difficulties 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 point in the CAN volume. This creates a problem for the one-to-one mapping of nodes to zones. Additionally, many jobs might have very similar requirements. For example, many jobs will likely be inserted into the system with no resource requirements at all specified. In this case, all those jobs are mapped to the single node that owns the corresponding zone. We address this problem by augmenting both job and node descriptions with a randomly assigned value in a "virtual" dimension. The virtual dimension ensures that all jobs and nodes are unique, and helps balance load even when the actual jobs and nodes are similar.

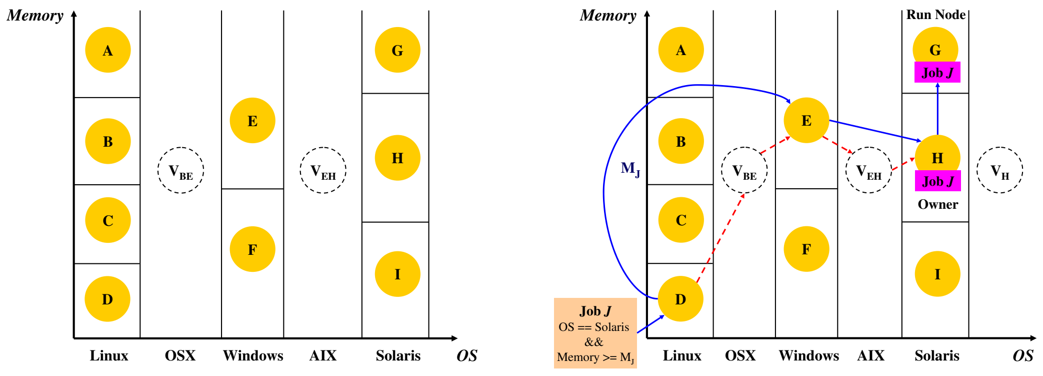
Second, we improve load balancing further by *pushing* jobs into underloaded regions of the CAN space. Nodes periodically send load information towards the origin in each dimension. This information is aggregated at each step, resulting in each node having partial information about load in all regions of the CAN space containing nodes more capable, which are exactly those nodes that are also able to run that node's jobs. In times of high load, a node can therefore *push* jobs towards regions of high capability and low load, based completely on local information. More details about our basic framework for job executions in a P2P network can be found in Kim et. al. [6, 7]

3 Integration of All Resources in a CAN

In this section we describe the details of our new techniques to integrate all types of resources (continuous and categorical) into a single CAN space.

Dealing with continuous and categorical resource types is not trivial. The system must be able to search for exact matches for the categorical resource types and minimum matches for the continuous resource types, while balancing load among multiple candidate nodes. One example of a possible user query for a set of required resources is (Arch == "Intel" $\wedge$ OS == "Linux" $\wedge$ CPU $\geq$ 2.4GHz $\wedge$ Memory $\geq$ 500MB $\wedge$ Disk $\geq$ 1GB), where Arch and OS are the required processor architecture and operating system type, respectively. To be able to handle this kind of query, the system has to find nodes that both have an Intel architecture and the Linux operating system, and also that meet the remaining continuous resource constraints (i.e., CPU, Memory and Disk).

One straightforward approach to integrate different types of resources into a CAN space would be to add new dimensions for categorical resource types (e.g., a dimension for



(a) Resource Integration (b) Routing including Virtual Peers

Figure 1. Resource Integration and Routing in a CAN space: In Figure 1(b), solid arrows denote the physical routing path of job J, while dotted arrows show the logical routing path.

architecture and a dimension for operating system in the example). The primary problem with this approach is in specifying the load information that must be aggregated and disseminated throughout the system to perform load balancing. The load information must distinguish between machines with different architectures (e.g., Intel and PowerPC), and also between different operating systems (e.g., Linux and Windows). Moreover, load information must be differentiated on the basis of all *combinations* of these choices; the number of such combinations is exponential in the number of discrete choices for each categorical resource type. A second approach is to create a distinct CAN space for each such combination of choices for categorical resource types. Load information within each such sub-CAN is then homogeneous and can be disseminated efficiently. The drawback of this approach is that such a system requires some type of directory service that vectors incoming jobs to the correct sub-CAN and manages the multiple sub-CANs. This frontend is both a potential performance bottleneck, and also a single point of failure.

Our solution is to integrate categorical resource dimensions into a single CAN space, by **transforming** *them onto a single dimension using a space-filling curve [10]. Then, we address load balancing and connectivity issues by introducing* **virtual peers**.

Figure 1 shows the basic concepts of our approach for integrating categorical resource types into a CAN (as an example, we use operating system for the categorical resource type and memory for the continuous resource type). The basic idea of the approach is to divide the CAN space into multiple *disjoint sub-spaces* where in each sub-space all of the categorical resource types are exactly the *same*, and provide an efficient mechanism to *connect* the multiple sub-spaces (without having a directory service). For example, in Figure 1(a) all nodes in the "Linux" range (A, B, C, and D) have the Linux OS. There is no node that has another operating system type (such as Windows) in that sub-space. Similarly, nodes G, H, and I have the Solaris OS. The overall CAN space is thus divided into three different sub-spaces, for Linux, Windows and Solaris. The question then is what happens to the rest of the CAN space (i.e., the sub-spaces for OSX and AIX). The OSX and AIX sub-spaces are *empty* because no nodes have those OS types. Therefore, there can be *holes* in the CAN space, since a sub-space of the CAN is occupied only if there is at least one real node that has that categorical resource type. However, we cannot just allow holes in the CAN space since they may prevent routing requests from being delivered.

We address this problem by supplementing the *physical* peers with additional *virtual* peers, as shown in Figure 1(a), where the OSX and AIX sub-spaces are occupied by two virtual peers V_{BE} and V_{EH}, respectively. Virtual peers act similarly to physical peers, both maintaining neighbor information and allowed to be neighbors of physical peers. However, a virtual peer never is allowed to become a neighbor of another virtual peer, since a single virtual peer can cover multiple *unoccupied* CAN sub-spaces. Since a virtual peer is not a physical node, we provide a mechanism to *map* each virtual peer to physical peers (called *manager* nodes). A manager node of a virtual peer maintains all information about the virtual peer (e.g., neighbor list) and processes any routing requests for its assigned virtual peer(s). In Figure 1(a), V_{BE} is managed by nodes B and E while V_{EH} is mapped to nodes E and H. A virtual peer can be managed by up to *two* different physical peers (the number of mapped physical peers depends on whether the virtual peer is an edge or an internal virtual peer in the integrated CAN space), enabling robust failure recovery.

With this design, each physical peer only is responsible for the exact region of the CAN space to which it belongs, with respect to its categorical resource specifications, and the rest of the space is covered by virtual peers. This enables employing the efficient matchmaking and load balancing techniques presented in Section 2.1, since in each sub-space the existing algorithms can aggregate the load information along the *continuous* dimensions, and employ the job pushing mechanisms for better load balancing within a single CAN sub-space, without considering different types of categorical resources. Figure 1(b) shows the overall procedure of matching a job *J* to node G, showing both physical and virtual peers in the CAN space. Since each virtual peer is mapped to one or two physical peers, a job request can be efficiently delivered to the owner node, as shown in Figure 1(b) (e.g., when node D routes the job request to the virtual peer V_{BE}, it can directly send the job to the physical peer E that V_{BE} is mapped to).

3.1 1-Dimensional Transformation

In Figure 1, we show only a single categorical resource dimension. However, the management of virtual peers and failure recovery mechanisms can become very complex with multiple categorical dimensions. The number of empty *sub-spaces*, each requiring a virtual peer, can increase combinatorially with the number of categorical dimensions.

To address these problems, we *transform* all categorical resource types into a *single* dimension. The overall CAN space is then composed of one transformed categorical resource dimension (we call this dimension *T*), along with all other continuous resource dimensions (including the virtual dimension described in Section 2.1). Any type of 1-dimensional transformation function can theoretically be used for this purpose, but consider that a user query may specify "don't care" or a limited range query as the requirement for a categorical resource type. In that scenario, the resource query specified for a job becomes a range query in a multi-dimensional space, so that a simple transformation function, such as a row-major or a column-major ordering, favors one dimension over others.

To support range queries, we preserve locality by using a Hilbert space filling curve (HSFC) [10] for the transformation. A HSFC is a continuous mapping from a *d*-dimensional space to a 1-dimensional space, passing through every point in a *d*-dimensional space exactly once, resulting in an ordering with good locality properties across all dimensions.

Transforming all of the categorical resource types into a single dimension allows us to efficiently introduce virtual peers to cover gaps in the CAN space. A single dimension allows a contiguous set of missing configurations (that have no real peer with those values for the categorical dimensions) to be represented with a single virtual peer. The number of virtual peers then grows only linearly with the number of *existing* configurations.

With a single transformed *T* dimension, coordinates for the nodes and jobs are generated by transforming categorical resource specifications (constraints) and combining that with continuous resource capabilities (constraints). We modify the node join algorithm so that if a new node splits one of the existing nodes in the system, our previous zone splitting algorithm will be applied. However, if the new node is mapped to a virtual peer's zone, it becomes the first node for the sub-CAN determined by its categorical resource types and becomes the manager of the new virtual peers. We also modify the node leave (failure recovery) algorithm so that a virtual peer takes over a recovered zone along the *T* dimension only if the *last* node in a sub-CAN departs the system (or fails). Finally, if a job is routed to one of the virtual peers in the matchmaking process, this means that there is no node in the system that can meet the job's resource constraints, so the job cannot be run.

4 Improving Scalability

Although the design in Section 3 can effectively match incoming jobs with various types of resource constraints to available resources, it has several drawbacks for system maintenance that we now describe.

As shown in Figure 1, a virtual peer becomes the neighbor of *all* physical peers that abut in the *T* dimension (for example, V_{BE} is the neighbor of nodes A, B, C, D, E, and F). This means that a virtual peer must exchange heartbeat messages with many neighbors periodically, and the size of each message grows with the number of the virtual peer's neighbors (this is because each node in the CAN sends its own information and its neighbor information in a periodic update message [9]). Such messages can add substantial overhead for the nodes responsible for the virtual peers. We evaluate the overhead from periodic update message exchanges in detail in Section 5.

A similar problem can occur with the continuous dimensions. Specifically, whenever there are sets of homogeneous computational resources in the system (e.g., a workstation cluster), some nodes might have many neighbors along the *virtual dimension* from the zone splitting process. However, unlike the virtual peer case, this does not always happen since it depends on the order of nodes joining. This problem occurs because, unlike the original CAN DHT, our CAN has dimensions with *semantics* corresponding to resource types. Therefore, we cannot guarantee that a zone for a new node will be split along a specific dimension, because the resource capabilities of nodes are not truly heterogeneous.

Another potential problem with the virtual peers is that some nodes may process more routing messages than other nodes. For example, in Figure 1, a job that requires the Linux operating system type may start from node H (which is the injection node of this job), where only Solaris machines are located. That job must traverse multiple sub-CANs until it arrives at the right sub-CAN (in terms of categorical resource types). In this step of matchmaking, the manager nodes will be used for routing jobs across sub-CANs, since the only way to traverse sub-CANs is through the virtual peers. Therefore, the manager nodes can end up routing a large number of matchmaking messages.

All of these issues can limit system scalability, as they complicate system maintenance with increasing numbers of nodes and jobs. In the rest of this section we describe new techniques to address all these problems, to achieve good scalability as shown in Section 5.

4.1 Modified Heartbeat Messaging

To deal with large periodic update messages, we introduce a *partial update* mechanism for heartbeat messaging. Whenever a node sends information about its neighbors, it may only send *partial* neighbor information. For this purpose, we use a threshold value, *PU_Threshold*, which limits the number of neighbors that are included in a periodic update message in each *direction* (upper or lower in each dimension). Therefore, even with only partial information about neighbors, each node will let its neighbors know about at least one and at most PU_Threshold neighbors in each direction (we select the PU_Threshold neighbors from each direction *randomly*). This ensures the correctness of the failure recovery algorithms, so that whenever a node leaves the system or fails, the neighboring nodes can determine the neighbors of the lost zone through the neighbor information maintained in that direction. Note that the partial updates mechanism does not affect failure recovery along the *T* dimension, since a physical peer never takes over the zone along that dimension (only virtual peers do).

In addition to the partial update mechanism, to reduce the overhead of message exchanges between a virtual peer and its neighbors, we also employ *probabilistic* heartbeat messaging. As discussed earlier, the node takeover operation along the *T* dimension occurs only if the last node in a sub-CAN departs. Therefore, all physical peers abutting a virtual peer do not have to send heartbeat messages to the virtual peer (except the ones that manage the virtual peer). This reduces the number of incoming messages to the virtual peer. Also, the virtual peer limits how often it sends heartbeat messages to any given neighbor through the partial update mechanism described previously, which only lengthens the average time between heartbeat messages sent to each neighbor.

4.2 Routing in the T Dimension

To address the problem of hot spots for processing routing messages from one sub-CAN to another sub-CAN, we use a special routing mechanism in the *T* dimension.

Whenever a physical peer tries to route a request to the virtual peer, it sends the request to one of the *neighbors* of the virtual peer (rather than sending directly to the manager of the virtual peer). Therefore, in the *T* dimension, the algorithm utilizes the neighbor of neighbor information maintained by the CAN, and routing requests are processed not only through direct neighbors but also *indirect* neighbors. For example, in Figure 1(b), when node D routes the request for job *J*, it selects one of V_{BE}'s neighbors (node E or F) and sends the request. This prevents all routing requests delivered from the Linux sub-CAN to another sub-CAN from always going through node E, the manager of V_{BE}.

5 Evaluation

In this section we evaluate our decentralized algorithms through a comparative analysis of experimental results obtained via simulations.

5.1 Experimental Setup

We use synthetic job and resource (node) mixes to simulate the behavior and measure the performance of our proposed CAN-based matchmaking algorithms. The resource mixes are modeled after common environments the system will run in (a combination of workstation clusters and desktop machines), and from a variety of job mixes obtained from our astronomy collaborators. 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).

We used five different resource types for nodes and jobs: CPU architecture, operating system type, CPU speed, memory, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used *two* different combinations: (Intel/Linux) and (AMD/Linux). So nodes and jobs can belong to either of the combinations with respect to their resource specifications and constraints, respectively. This is representative of heterogeneous cases where more than two combinations of the categorical resource types exist, but still shows the basic behavior of the CAN. Also, this case will show how having virtual peers with a large number of physical neighbors affects overall system performance.

We generate the continuous resource values (CPU, memory and disk) for nodes and jobs based on a *clustering model*, as described in our earlier work. The clustering model emulates the 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 [12]). We used ten different sets of homogeneous clusters having different amounts of continuous resource capabilities. As described in our previous work [6, 7], our algorithms can also handle truly heterogeneous set of nodes and jobs where there are few identical nodes (an *unclustered* set of nodes). However, we use the clustered model for our experiments since this is a likely scenario and it also shows the behavior of the system when nodes have many neighbors along the virtual dimension, as described in Section 4.

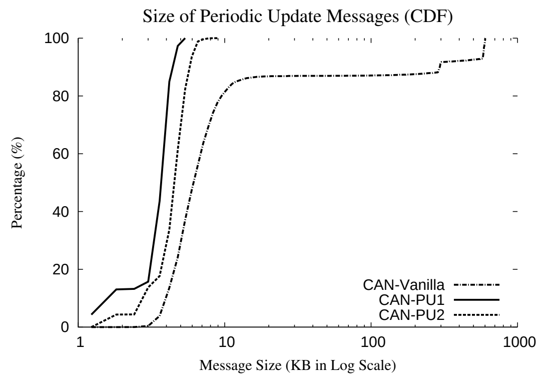
The amount of work *W* for a job *j* is generated uniformly at random from a predefined set of work ranges (60 minutes on average), and means that to run the job *j* a node must execute for *W* time units if it has *exactly* the same node specification as does the job *j*'s constraints. To model the 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 network communication cost we model the latency of a packet between any two nodes by an exponential distribution with a mean of 50 milliseconds.

One important characteristic of the test workload is that the overall system reaches a *steady state*, for both available nodes and active jobs, during the simulation period. The way we generated the workload is that first an initial set of 1000 nodes join the system. Then, new node join events and existing node departure events (graceful leaving or failure) occur at approximately the same rate. Once the system reaches the steady state in terms of active nodes a total of 5000 jobs are submitted. Again, the system behavior is measured in a steady state, so that the number of active jobs remains about the same (jobs arrive and complete at about the same rate), and we 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.

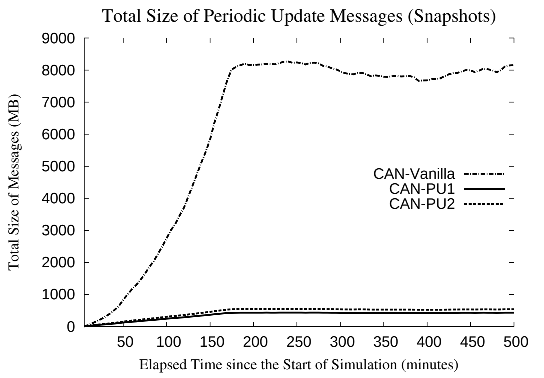
Our metrics are *matchmaking cost* (the amount of time between when a job is injected and when it is assigned to a run node) (**MC**) and *queuing time* (the amount of time between when a job is inserted into a run node and when it actually starts running) (**QT**). MC directly quantifies the overhead needed to perform the matchmaking in a decentralized manner. QT includes the time spent waiting in the job queue of a run node before a job is executed (i.e., indirectly measuring load balance).

We test the CAN approach both before addressing scalability issues (Section 3) (**CAN-Vanilla**) and the improved CAN approach employing partial updates (described in Section 4.1) and the special routing algorithm (Section 4.2). For partial updates, we used two different PU_Thresholds, 1 and 2 (**CAN-PU1** and **CAN-PU2**, respectively, see Section 4.1). Since both thresholds showed similar (good) behavior, we do not show results for higher threshold values. To see how well the workload could be balanced, we also show results for a centralized scheme (**CENTRAL**) that uses knowledge of the status of all nodes and jobs. Such a scheme would be very expensive to implement with a distributed set of nodes, but serves as a target for achieving the best possible load balance from an online matchmaking algorithm. The matchmaking cost for the centralized approach is 0.

5.2 Results for System Maintenance



(a) Partial updates reduce many message sizes by orders of magnitude



(b) Total bandwidth consumption with partial updates drops by a factor of 10

Figure 2. Size of Periodic Heartbeat Messages Exchanged

Figures 2 and 3 show the behavior of our algorithms with the scalability improvement techniques. With the use of partial updates, we can greatly reduce the size of each periodic update message, as seen from Figure 2(a) (note that the x-axis is *log scale*). Compared to CAN-Vanilla, CAN-PU1 and CAN-PU2 are sending very small heartbeat messages. The main reason for CAN-Vanilla having larger message sizes is that the virtual peers have a large number of neighbors, and send all that neighbor information in heartbeat messages. This problem is amplified since each node periodically (every 30 seconds for our simulations) sends heartbeat messages to all its neighbors. Figure 2(b) shows the total size of the heartbeat messages sent between all nodes during every 5 minute interval, and we see that CAN-Vanilla sends a much larger amount of data than either CAN-PU1 or CAN-PU2.

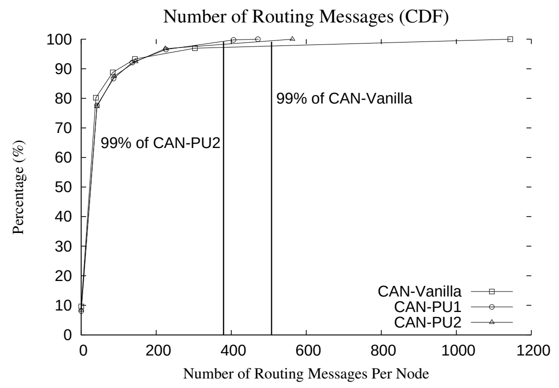


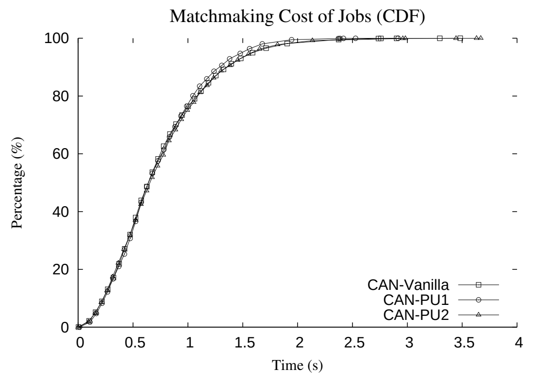
Figure 3. The T-dimension routing algorithm reduces the message burden on manager nodes compared to CAN-Vanilla.

The partial update mechanism may affect failure recovery along the continuous dimensions, since each node has only a small amount of information about the neighbors of its neighbors. So when an existing node leaves the system or fails, the node that takes over that zone, call it *N*, might not be able to find all new neighbors abutting the lost zone (since the departed node did not provide node *N* all of its neighbor information). Therefore, temporary holes that no node owns can occur in the CAN space, since a node that takes over a zone may not have complete neighbor information after merging the lost zone with its own zone. However, these holes are quickly repaired through later heartbeat messages when neighbor information is exchanged, since the partial update algorithm always sends information about at least one neighbor in each direction in the periodic update messages. In experiments not shown, we verified that the average time to recover from failures along the continuous dimensions with partial updates is only a factor of three worse compared to sending complete updates (CAN-Vanilla takes an average of around 20 seconds to repair holes).

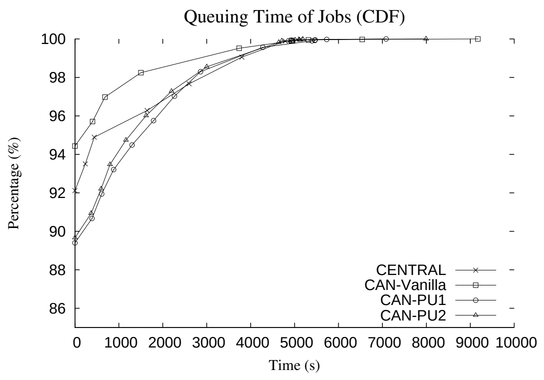
In addition to the partial update mechanism, we proposed a special routing algorithm in the *T* dimension to avoid hot spots for routing requests. The results are shown in Figure 3, which shows the number of routing messages processed by each node during the entire simulation. The figure shows that CAN-Vanilla, which does not use the special routing algorithm in the *T* dimension, shows the worst performance in terms of number of routing messages processed. The main reason for this is that the physical peers maintaining the virtual peers (the manager nodes) must process a large number of routing messages to deliver requests between the Intel/Linux and AMD/Linux sub-CANs.

5.3. Results for Job Executions

Figure 4 shows the matchmaking cost and queuing time of jobs with the different algorithms. As we can see from



(a) Matchmaking is not noticeably impacted by partial updates



(b) Stale queue information for neighbors results in slightly longer queuing times with partial updates

Figure 4. Matchmaking and Queuing Time

Figure 4(a), the CAN-based mechanisms can match the jobs having different resource constraints with available heterogeneous resources with very low cost. Most of the jobs can be matched within a couple of seconds, but some take much longer. High matchmaking costs can occur when new nodes join or existing nodes leave the system Those events cause transient system states where matchmaking for a job cannot proceed until the holes in the CAN space caused by the node departures have been repaired. During those periods CAN routing can fail, so has to be retried. However, jobs are still matched within a very short time period, since the system quickly recovers from those transient states.

Figure 4(b) shows the quality of load balancing of the CAN-based approaches compared to the centralized matchmaker. As the figure shows, all of the CAN-based frameworks show performance competitive with CENTRAL, although there are some jobs that wait longer in the queues. We showed in our previous work that in this scenario, with sets of homogeneous clusters of nodes, the CAN performs well, both in load balancing and matchmaking cost, because splitting the CAN in the virtual dimension for clusters of homogeneous nodes helps spread jobs with similar resource requirements across all nodes in a cluster capable of running the jobs [6, 7]. In addition, through the experiments shown here, we verify that the partial update mechanism does not affect the quality of load balancing very much, so that our algorithms perform well for both categorical and continuous resource capabilities for nodes and requirements for jobs.

6 Related Work

Research such as [3, 8] employed a *Time-To-Live* (TTL) mechanism to locate and allocate resources in a Grid environment. TTL-based mechanisms are relatively simple and effective ways to find a resource that meets the job requirements, but such mechanisms may fail to find a resource even though one exists somewhere in the network.

Similar to our approach, research such as [4, 5] encoded static or dynamic information about computational resources using a DHT hash function for resource discovery. However, 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. Also, simple encoding of resource information cannot effectively avoid selecting resources that are over-provisioned with respect to the jobs.

A peer-based desktop grid system called WaveGrid [12] constructed a *timezone-aware* overlay network based on a CAN to use idle night-time cycles geographically distributed across the globe. Balanced Overlay Networks (BON) [2] encode information about each node's available computational resources, resulting in a self-organized network that allows jobs to be assigned to free nodes via short random-walks. However, the job allocation model in those systems does not consider the requirements of the jobs nor the varying resource capabilities of the nodes.

7 Conclusions and Future Work

In this paper, we have described a P2P desktop grid system that can efficiently match exact and minimum requirements for jobs simultaneously, while balancing load among multiple candidate nodes. By introducing virtual peers and a 1-dimensional transformation using a space-filling curve, we have integrated different types of resources into a single CAN. However, this architecture can scale poorly because of asymmetric peering in the CAN space. We described several novel techniques that greatly improve scalability by optimizing the information flow, as well as routing across category boundaries. The resulting decentralized system is both scalable and efficient in performing matchmaking for complex jobs and environments that mix continuous and categorical resources.

We are in the process of implementing our system, and will characterize its behavior on real workloads, in cooperation with our collaborators in physics and astronomy. In the near future, we will measure and report on the behavior of our system for heterogeneous environments running real applications.

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] J. Bridgewater, P. O. Boykin, and V. Roychowdhury. Balanced Overlay Networks (BON): An Overlay Technology for Decentralized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):1122–1133, 2007.

[3] D. Caromel, A. di Costanzo, and C. Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, 33(4-5):275–288, 2007.

[4] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer Discovery of Computational Resources for Grid Applications. In *Proceedings of GRID 2005*, Nov. 2005.

[5] 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.

[6] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids. In *Proceedings of HPDC 2007*, June 2007.

[7] 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 GRID 2006*, Sept. 2006.

[8] 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 (EGC)*, Feb. 2005.

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

[10] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, 2006.

[11] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-Peer resource discovery in Grids: Models and systems. *Future Generation Computer Systems*, 23(7):864–878, Aug. 2007.

[12] D. Zhou and V. Lo. WaveGrid: a Scalable Fast-turnaround Heterogeneous Peer-based Desktop Grid System. In *Proceedings of IPDPS 2006*, Apr. 2006.