Autonomic Load Balancing Mechanisms in the P2P Desktop Grid

Jik-Soo Kim
National Institute of Supercomputing and Networking
Korea Institute of Science and Technology Information
Daejeon, Republic of Korea
jiksoo.kim@kisti.re.kr

Beomseok Nam
School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology
Ulsan, Republic of Korea
bsnam@unist.ac.kr

Alan Sussman
Department of Computer Science
University of Maryland
College Park, MD, U.S.A.
als@cs.umd.edu

ABSTRACT

Peer-to-Peer (P2P) desktop grid computing systems circumvent the performance bottleneck and limited scalability of centralized Grid architectures resulting in a massively scalable and robust system. We have designed a set of protocols that implement a distributed, decentralized desktop grid via P2P techniques. Incoming jobs having different types of resource requirements are matched with system nodes through proximity in an *N*-dimensional resource space.

In this paper, we address problems that arise from static load balancing mechanisms for assigning jobs to nodes that can arise for various reasons, including the heterogeneity of the available nodes or the jobs to be run, and from stale information in the P2P system. We greatly improve upon our prior work by providing lightweight yet effective dynamic load balancing mechanisms to overcome load imbalances caused by the limitations of the initial static job assignment scheme. Unlike other systems, we can effectively support *resource constraints* of jobs during the course of redistribution since we simplify the problem of matchmaking through building a multi-dimensional resource space and mapping jobs and nodes to this space. Throughout extensive simulation results, we show that dynamic load balancing makes the overall system more scalable, by improving system throughput and response time with low additional overhead.

Keywords

Desktop Grid Computing, Peer-to-Peer System, Autonomic Load Balancing, Resource Constraints

1. INTRODUCTION

Peer-to-Peer (P2P) desktop grid computing systems circumvent the performance bottlenecks and limited scalability of centralized Grid architectures [18, 31] such as Condor [30] or BOINC [1]. Conventionally, these kinds of middleware systems have effectively supported High-Throughput Computing (HTC) [19] consisting of running many loosely-coupled tasks that are independent (there is no communication needed between them) but require a large amount of computing power during relatively a long period of time. However, as the number of jobs and the complexity of scientific applications increase, it becomes a challenge to solve the given scientific problem within a reasonable amount of time. Also, recent emerging applications requiring millions or even billions of tasks to be processed with relatively short per task execution times have led the traditional HTC to expand into *Many-Task Computing* (MTC) [22]. These applications from a wide range of scientific domains (e.g., astronomy, physics, pharmaceuticals, chemistry, etc.) often require a very large number of tasks (from tens of thousands to billions of tasks), and have a large variance of task execution times (from hundreds of milliseconds to hours) [22].

Therefore, to effectively support most challenging scientific applications (from HTC to MTC), first, we need to build a system that can scale out to harness hundreds of thousands of computing resources to support millions or even billions of tasks. Second, such a system must be robust enough to provide high availability to the scientific users who want to process a large number of jobs across multiple resources (no single point of failure). Third, the system must incorporate the heterogeneity in the running times of jobs and in the resource capabilities of the nodes so that it can dynamically adjust to the load distribution changes.

We believe that the P2P desktop grid can be a viable choice for such a scalable and robust system that can process a tremendous number of tasks by harnessing vast amounts of computing resources. In this paper, we present our lightweight and effective *autonomic load balancing* mechanisms that can redistribute jobs in a completely decentralized fashion incorporating job constraints, and address the load imbalance issues arising from the heterogeneity in the execution times of jobs and in the resource capabilities of the nodes. In our P2P desktop grid system [15, 16, 17], matching jobs to resources (*matchmaking*) are based on proximity between job and node characterizations in an *N*-dimensional Content-Addressable Network (CAN) [25]. In that approach, each resource type corresponds to a distinct dimension.

Unlike other P2P-based resource management schemes [11,

CAC '13 August 05-09 2013, Miami, FL, USA

28, 33, 34], we can effectively support *resource constraints* of jobs during the course of redistribution since we simplify the problem of matchmaking through building a multi-dimensional resource space and mapping jobs and nodes to this space. As the number of various scientific applications increases, with high probability, it will be crucial to support resource requirements of running jobs on top of heterogeneous computing infrastructures. Also, our system can give an insight to decentralize existing job scheduling systems such as Falkon [23, 24] or MyCluster [32] by effectively building a self-organizing pool of schedulers connected via a P2P network which can provide more robust and scalable dynamic load balancing mechanisms.

The rest of the paper is structured as follows. Section 2 describes our goals and prior work for matching jobs to resources based on CAN. In Section 3, we describe our techniques to improve the overall throughput of our CAN-based system by employing dynamic load balancing schemes and present our simulation results in Section 4. Section 5 presents related work and we conclude and discuss future directions in Section 6.

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 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 System Overview

Our system [15, 16, 17] is based on a distributed hash table (DHT) called Content-Addressable Network (CAN) [25] and we found that it provides a good framework for a decentralized desktop grid. A CAN is a type of structured 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 (constraints). 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. This is a typical scenario in a high-throughput computing environment, enabling many independent users to submit their jobs to a collection of node resources in the system, or *embarrassingly parallel* (sometimes called *bags of tasks*) workloads. Indeed, Iosup et al. [13, 14] found that a high percent of Grid applications still employ an embarrassingly parallel model based on their analysis on the characteristics of traces of real Grid environments.

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 initiate selecting 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* [16]. 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 for executing jobs by *pushing* jobs into underloaded regions of the CAN space [15]. Nodes periodically send load information towards the origin in each CAN 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,– exactly those nodes that are also able to run the node's jobs. In times of high load, a node can therefore push jobs towards regions of high capability and low load, based solely on local information.

We also integrate *categorical* resource constraints [17] which 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 (e.g., memory or disk size, or CPU speed). The basic idea of our approach is to divide the CAN space into multiple *disjoint sub-spaces* where in each sub-space all categorical resource types are exactly the same, and then provide an efficient mechanism to connect the multiple sub-spaces through

virtual peers. 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 (unoccupied spaces) is covered by virtual peers. Since a virtual peer is not a physical node, we map each virtual peer to physical peers (called *manager* nodes). For efficient management of virtual peers and failure recovery, we *transform* all categorical resource types into a *single* dimension using a Hilbert Space-Filling Curve [27]. Load information within each sub-CAN is then homogeneous and can be disseminated efficiently, so that matchmaking is efficient for *any* combination of categorical and continuous constraints.

3. DYNAMIC LOAD BALANCING

One way the CAN-based matchmaking techniques balance load across run nodes is through the use of randomly generated virtual dimension values for both node capabilities and job requirements, which acts to distribute clusters of nodes and jobs through the CAN space. They also use the job pushing mechanism during matchmaking to balance load across all nodes that are capable of running the job. However, all of our prior job load balancing mechanisms are based on a *static* allocation of jobs to nodes, and do not allow jobs to be migrated to run on another node after it has been assigned to an initial run node.

Static load balancing has drawbacks, both because of *heterogeneity* in the running times of jobs and in the resource capabilities of the nodes. Even if the load balancing mechanism initially assigns the jobs uniformly across available system resources, as time passes the overall load distribution may change because some nodes run the allocated jobs much faster than others (or some jobs just have relatively short running times). Therefore, the overall throughput of the entire system may heavily depend on its slowest nodes. Also, we use the *number of jobs* in the queue at a run node as the metric to determine the best run node for a job when there are multiple candidates capable of running the job. This is because it can be very difficult in general to predict the actual running time of a job on a given node, unless clients provide such information and it is accurate for all node types in the system. However, the actual queuing time for a job is not necessarily directly proportional to the number of jobs in the queue, since the job running times can vary widely [22]. A final source of uncertainty comes from the decentralized nature of the P2P desktop grid system. All matchmaking and load balancing decisions are made based on only *local* information that is propagated over time as part of the basic CAN DHT maintenance algorithms. Therefore, if jobs are arriving faster than load information propagates, many matchmaking decisions will be made based on *stale* load information, which can result in load imbalances across run nodes.

To address these problems, we have designed *autonomic and dynamic load balancing mechanisms* that can effectively redistribute the jobs (with resource constraints) across run nodes as needed, to improve overall system throughput. However, job redistribution (migration) has both benefits and costs. Job migration cost may be higher in a P2P system that spans a wide-area network compared to a local area network, since the job profile has to be transferred, including all input data. For jobs that do not run for a long time, the migration cost may be very high compared to the job execution time. Jobs that run for hours or days can greatly benefit from migration, rather than sitting in a queue for a long time. Therefore, long running jobs having minimal data communication cost are most appropriate for job migration. Most long running desktop grid applications, such as those performed by SETI@Home [3] or Folding@Home [10], are indeed long running [33] and are the main target applications for such systems. However, for those applications generating a very large number of jobs with relatively short per task execution times (as in MTC applications [22]), our dynamic load balancing schemes can still work since many tasks will be waiting for execution due to lack of enough resources to support them and a large variance in task execution times will inevitably skew the overall load distribution.

We choose to only employ the job migration techniques to jobs that are not currently running, but are currently *waiting* in a queue on a run node, since migrating running jobs requires complex mechanisms for state storage and resuming the job. Also, in our system, any input data files for a job are transferred to the run node only when the job actually starts running. This means that by targeting only jobs waiting in the queue, the job migration cost is low since we only need to migrate the job description file, which is quite small. We now present our dynamic load balancing schemes, based on either *pulling* jobs to lightly loaded node or *pushing* jobs away from heavily loaded nodes.

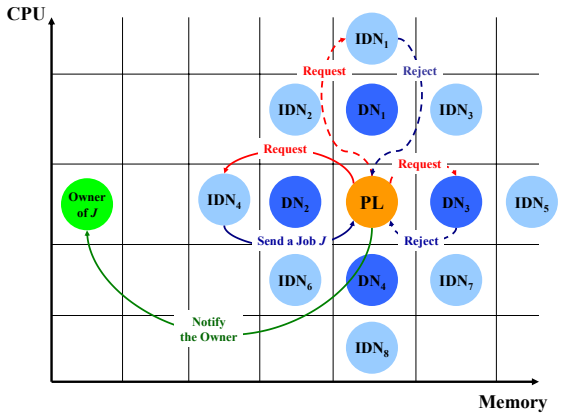
3.1 Models for Migrating Jobs

In the push model, a node that has a disproportionate number of jobs in its queue can *push* jobs to its neighboring nodes in the CAN, while in the pull model a node that becomes idle can *pull* jobs from its more heavily loaded neighbor nodes. However, the semantics of the matchmaking process and the CAN organization can make this procedure difficult, since **we must ensure that a node receiving a migrated job meets the resource requirements of the job**. Also, it is desirable to perform job redistribution in a completely decentralized, local fashion to avoid multiple retrials of the entire matchmaking process just to migrate jobs.

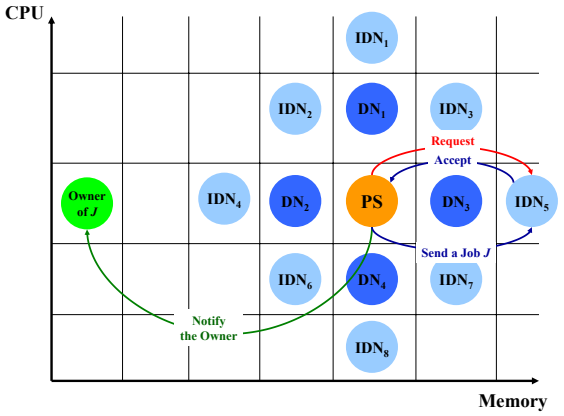
Decentralized dynamic load balancing can be done using the neighbor state information that must be maintained for connectivity in the CAN space. From the perspective of a node, its neighbor nodes are good candidates for running jobs in its queue, since they are likely to meet the constraints of those jobs, due to the assignment of resource types to the different CAN dimensions. Periodically, and independently of when other nodes send updates, a node sends its own current information (such as zone, coordinates, etc.) and the same information that it currently has for its neighbors in the CAN space to all its neighbors [25]. Therefore, each node maintains both the state of its *direct* neighbors and also state for neighbors of neighbors (*indirect* neighbors). This information is required to enable the basic CAN failure recovery mechanism, where the node that ends up taking over the zone vacated by a failed neighbor can discover the neighbors of the lost zone through its indirect neighbor information [25]. In our desktop grid CAN, additional load information (i.e., the current size of the job queue) is piggybacked onto the periodic neighbor updates so that each node can estimate the current load of its direct and indirect neighbors.

Based on load information about its neighbors, a node pe-

riodically performs dynamic load balancing, but at a longer interval than for updating the neighbor state information. That is because job redistribution should not add substantial overhead, and also because the system targets jobs that usually run long enough so that relatively infrequent job redistribution will be adequate to smooth out any load imbalances caused by the static load balancing scheme and widely varying job run times.



(a) Pull Model



(b) Push Model

Figure 1: Models for Dynamic Load balancing: DN_i and IDN_i denote direct and indirect neighbors, respectively.

PULL Model.

Figure 1 shows two different approaches for the dynamic load balancing of jobs, a pull model (Figure 1a) and a push model (Figure 1b). In the pull model, whenever a node *PL* becomes idle (or very lightly loaded), it tries to pull jobs from its more heavily loaded neighbors (both direct and indirect). In Figure 1a, node PL is performing dynamic load balancing and has twelve neighbors that can be considered. PL sorts those neighbors according to their job queue sizes (propagated through the neighbor updates), and selects the one that has the longest job queue size (but it must be longer than PL's job queue). PL then sends a message to that neighbor to request a job (*Request* in the Figure 1a).

One important constraint is that PL must be able to run the job migrated from its neighbor (i.e., it should meet the resource requirements of the job). For that reason, one approach for the pull model is that PL contacts only neighbors that can be covered by its own coordinates (i.e., each coordinate of a neighbor is less than or equal to the corresponding coordinate of PL). For example, in Figure 1a, nodes DN_2, DN_4, IDN_4, IDN_6 and IDN_8 are covered by the coordinates of PL, which means that all jobs in these neighbors are *guaranteed* to be able to run on PL due to the semantics of the multi-dimensional CAN space. While this scheme guarantees meeting the job constraints, it restricts the flow of job migration to only one direction (always from regions closer to the CAN origin than PL). However, the static load balancing mechanism, which pushes jobs away from the node that has the minimum capability to run the job (i.e. meet its resource requirements), may push a job to nodes with higher resource capabilities [15]. That means that neighbors that cannot be covered by PL (the other 7 nodes in Figure 1a) may indeed have jobs that PL can run. Therefore, in our pull model node PL contacts one of its neighbors based only on their job queue sizes, ignoring their CAN coordinates. However, when the node PL contacts its most heavily loaded neighbor, that neighbor may not have any jobs waiting in its queue that can be run on PL. In this case, that neighbor simply sends a reject message to PL, and PL tries to pull a job from the next most heavily loaded neighbor node (*Reject* in Figure 1a). Although this may require several attempts to contact neighbors of PL, the overhead is not too high since the number of neighbors is limited and PL always contact more heavily loaded neighbors than itself (we investigate the overhead incurred by the pull model in Section 4).

Finally, if PL finds an appropriate node for migration, a job that can be run on PL (selected by searching from the head of the job queue on the node sending the job) is transferred to PL and inserted into PL's job queue (*Send* in Figure 1a). To ensure fairness among jobs in the system, we sort all jobs in a job queue based on their *submission times* so that migrated jobs may be inserted anywhere in a job queue, not just the end.

PUSH Model.

In the push model whenever a node *PS* becomes heavily loaded (i.e. its job queue gets long enough), the node attempts to push one or more of its queued jobs to its more lightly loaded neighbors (both direct and indirect), as seen in Figure 1b. PS sorts those neighbors according to their job queue sizes and picks the one that has the shortest queue size (and the job queue size is shorter than PS's). PS then contacts that neighbor to request job migration. However, unlike the pull model, node PS never has to make multiple attempts to find a neighbor that can run one of the waiting jobs in its queue. That is because all of nodes that are candidates for job migration from node PS are neighbors of PS, so PS can determine whether a neighbor can run the migrating job *before* making the request based on the job coordinates (resource requirements) all being less than or equal to the corresponding neighbor coordinates (resource capabilities). Therefore, when PS sorts its neighbors it can safely exclude nodes that cannot run any of the jobs in its queue. This keeps the number of messages the push model requires for performing dynamic load balancing mechanism low, as will be seen experimentally in Section 4.

Diffusion of Load.

In both the pull and push models, one important issue is

to determine the *idleness* of a node, since jobs should always migrate from heavily loaded nodes to idle ones. If a node N is free (no running or waiting jobs in its queue), we can definitely regard N as an idle node. Therefore, in this case, node N should *always* try have a job migrated from one of its neighbors (through either the pull or push mechanism). However, what should happen if N has only one or two jobs in its queue? Note that we calculate job queue size as the number of running *and* waiting jobs in the queue (e.g., a free node has a job queue size of zero). We could use a threshold to determine the idleness of node N so that if N has fewer jobs in its queue than the threshold it is regarded as an idle node. However, selecting a good threshold value that is independent of the job characteristics can be very difficult. Therefore, another possibility is for a node N to be regarded as idle if and only if it is free (zero queue length). So only free nodes will get jobs migrated from its more heavily loaded neighbors. However, this scheme also may not work well in the decentralized P2P grid environment, because a node shares migratable jobs only with its neighboring nodes. However, if only free nodes are allowed to migrate jobs, that will only balance load in the regions in the CAN space near free nodes, so that jobs may not be propagated over longer distances in the CAN. Therefore, even though all operations are performed locally, a better method would gradually propagate the effects of job migrations so that loads are *diffused* into the entire CAN space.

To achieve that behavior, we employ a *probabilistic* approach for each node to determine whether or not it will accept a migrated job from its neighbors. A node N accepts a job migration request with a probability of $\frac{1}{(1+N's\ job\ queue\ size)}$. Therefore, if a node N is free, it will always accept migrated jobs from its neighbors. Also, even if N has some jobs in its queue, it may get additional jobs migrated from more heavily loaded neighbors. This simple but effective scheme allows jobs to gradually move from heavily loaded regions to lightly loaded regions in the CAN space, resulting in global diffusion of loads across all available nodes. Note that for all job migrations, the new node must always meet the resource requirements of a migrated job.

Choosing the Best Node for Migration.

It is possible for a node N to receive multiple job migration requests from multiple neighbors at about the same time. For the pull model, that is most likely to occur at the locally most heavily loaded node, and for the push model the most lightly loaded neighbor is likely to have this problem. The solution is for N to decide which requesting node finally will get the job. The choice could be done randomly or in order of the requests, but we have designed a method tailored to the characteristics of the load balancing algorithms. For the pull model, if a node receives multiple requests for job migration, it selects the lightest loaded neighbor and sends a job to that node. For this purpose, whenever a node requests a job migration, it includes its current job queue size in the message. Similarly, for the push model if a node receives multiple requests for job migration, it selects its most heavily loaded neighbor. The final step of job migration is to notify the owner node for the migrated job about the migration, so that it can keep track of the the run node for the job (as shown in Figure 1).

4. EVALUATION

We use synthetic job and compute resource (node) mixes to simulate the behavior and measure the performance of our CAN-based P2P desktop grid system. The resource mixes are modeled after common environments the system runs in (a combination of workstation clusters and desktop machines), and from a variety of job mixes obtained from our astronomy collaborators. 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 size, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used four different combinations (sub-CANs). Nodes (total 1000 nodes) and jobs (total 5000 jobs) have one of those combinations for their resource specifications and constraints, respectively. We generate continuous resource type 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 resource types and a small fraction of nodes have larger values for their resource types (as in Zhou et al. [33]). We used ten different sets of homogeneous clusters having different continuous resource capabilities, and the resource requirements for jobs are also clustered (i.e., multiple jobs have similar or even identical requirements).

If we designate the the amount of work for a job j by W, then to run the job j a node must execute for W time units if it has *exactly* the same CPU speed as specified by the job j's minimum CPU requirement. To model the actual running time of a job on the node to which it is assigned, we divide W by the node CPU speed (relative to some baseline node CPU speed). 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.

In these experiments, we varied two values, τ for jobs and the *distribution* of job running times. We used two different τ values: 1 and 4 seconds respectively (denoted as heavy and light workloads). With τ set to 4 seconds, the overall system stays in a *steady state*, where the rate for incoming jobs and finishing jobs is approximately the same. However, if τ decreases to 1 second, the system becomes heavily loaded and will eventually saturate all available nodes, resulting in indefinite growth of the node job queues. That scenario shows the behavior of our algorithms for dynamic load balancing in a very heavily loaded environment, where the static matchmaking decisions may be made based on stale information. Also, we used two different distributions for job running times, uniform and normal. In the uniform model, a job running time is generated uniformly at random from between 30 and 90 minutes and an average of 60 minutes. We also tested the algorithms with normally distributed job running times, with a mean of 60 minutes and a standard deviation of 20 minutes. This scenario shows how the algorithms are affected by non-uniformity in job running times, and also shows the effects of situations where the number of jobs in a node's queue is not a good estimate

for the queuing time of a newly assigned job. However, due to the page limit, in this paper, we only present results of uniformly distributed job running times.

Our metrics are *wait time*, which is the amount of time between when a job is injected by a client and when it actually starts running, and the *rate of dynamic load balancing messaging*, which is the number of messages required to perform the dynamic load balancing scheme per minute. 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. Since the matchmaking cost in our system is very small compared to the job running time [15, 16, 17], the majority of wait time is composed of the queuing time. The number of dynamic load balancing messages shows the overhead for executing the job redistribution algorithms in a decentralized fashion.

We compare the basic CAN approach that only uses the static job load balancing scheme (labeled as **CAN-Vanilla** in the figures) with the improved CAN approach that uses dynamic load balancing either with the job pull model (**CAN-PULL**) or the job push model (**CAN-PUSH**). Both CAN-PULL and CAN-PUSH perform the dynamic load balancing algorithms every five minutes, which is much longer than the 30 second interval between neighbor updates for CAN maintenance. To see how well the dynamic load balancing schemes work, we also show results for a centralized scheme (**CENTRAL**) that has complete information about the job queue status of all nodes. Similar to our dynamic load balancing mechanisms, CENTRAL periodically redistributes jobs across all nodes in the system. 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.

4.1 Experimental Results

We discuss the results for uniform job running time distributions, seen in Figures 2a and 2b. The figures show that the dynamic load balancing schemes (CAN-PULL and CAN-PUSH) greatly improve load balance compared to CAN-Vanilla, and show very competitive performance even compared with CENTRAL. Both CAN-PULL and CAN-PUSH not only remove the high end of the wait time distribution compared to CAN-Vanilla, meaning that the longest waiting jobs wait much less, but also shift the CDF up and to the left, which means that they achieve better distribution of jobs across available nodes compared to CAN-Vanilla. However, under the heavy workloads shown in Figure 2b, all of the matchmaking frameworks show longer job wait times compared to the lighter workloads. More specifically, CAN-PULL and CAN-PUSH decrease the average job wait time to 23% and 36% that of CAN-Vanilla, respectively, while CENTRAL decreases that metric to 22% that of CAN-Vanilla. For the heavy workload, CAN-PULL, CAN-PUSH and CENTRAL decrease the average wait times to 60%, 68% and 44%, respectively, that of the average job wait time for CAN-Vanilla. Therefore, by employing dynamic load balancing mechanisms, we can improve load balance for executing jobs dramatically compared to CAN-Vanilla, and shows performance close to that of CENTRAL, which has a global view of the entire set of nodes.

All the benefits from dynamic load balancing come with additional cost (overhead), since redistribution of jobs occurs *periodically*. However, as we can see from Figure 2c and 2d, the networking requirements to perform dynamic load balancing are very low, totaling only a few messages per minute. The messages counted include all those needed to perform the dynamic load balancing algorithms, which include contacting neighbor nodes to request a job migration, actually migrating a job, and notifying the job owner node of the new run node (as described in Section 3.1). One important characteristic about Figure 2c and 2d is that the graphs show the number of messages over the *entire* simulation, which means that for *all* one minute intervals simulated, there was no node in the system that processed more than 12 dynamic load balancing messages (for CAN-PULL). When we measure the total size of the dynamic load balancing messages across all the workloads, CAN-PUSH and CAN-PULL send up to 300 bytes and 600 bytes per minute respectively. As was described in Section 3.1, CAN-PULL generates more messages than CAN-PUSH, since a node can perform multiple retrials to contact its neighbors to find a job that can be run on that node. In results not shown, we also measured the average number of messages sent in the CAN per minute during the entire simulation and CAN-PULL on average causes only 0.3% of all messages (meaning the vast majority come from other sources, including CAN maintenance and matchmaking), while CAN-PUSH causes only 0.2% of the total number of CAN messages.

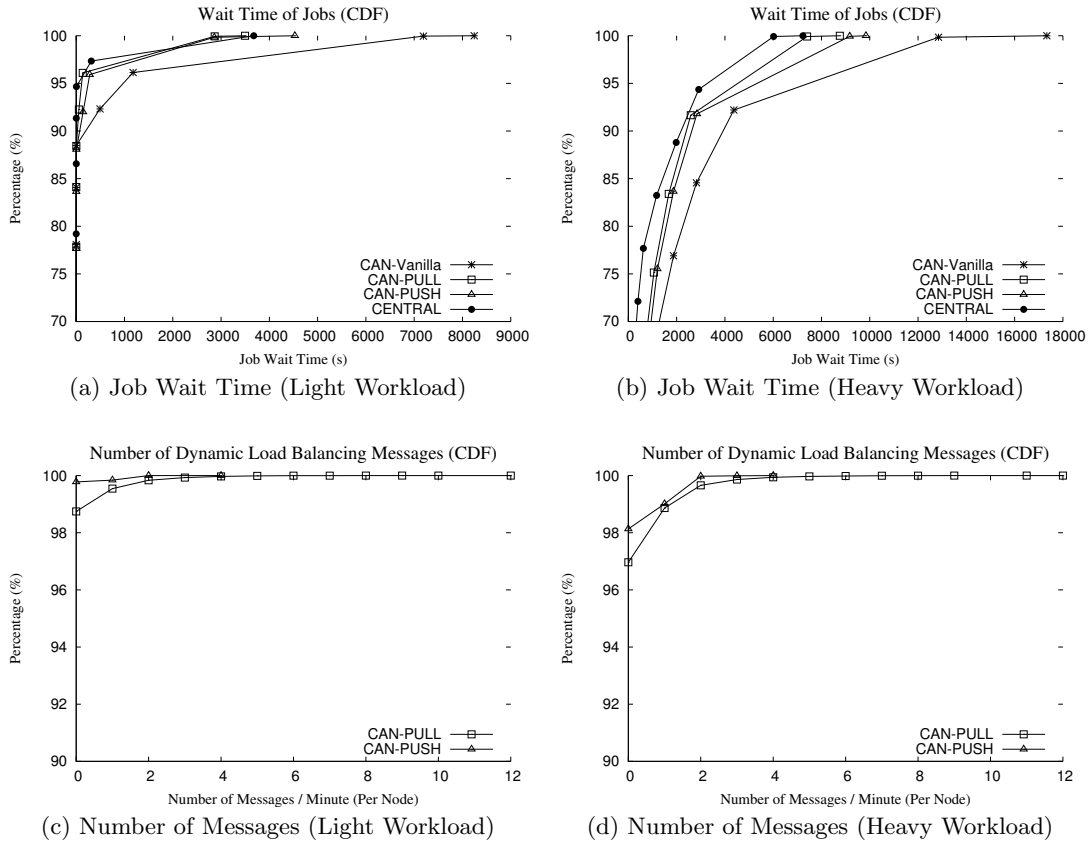
Another interesting result is that, across all of workload combinations (different loads and job running times), CAN-PULL provides better load balance (measure by job wait times) than CAN-PUSH. We believe that is because in CAN-PULL, idle (or comparatively lightly loaded) nodes aggressively pull jobs from their more heavily loaded neighbors, compared to the idle nodes in CAN-PUSH passively accepting jobs from their neighbors. However, since CAN-PUSH has the advantage of lighter overhead (counting messages) compared to CAN-PULL, both dynamic load balancing approaches have their strengths and weaknesses. So if the target system can handle some extra messages, CAN-PULL is the best choice. Otherwise CAN-PUSH should be used to perform dynamic load balancing, since it can still greatly improve overall system throughput compared to CAN-Vanilla, which does not do any dynamic load balancing.

5. RELATED WORK

Research such as [8, 20] employs a *Time-To-Live* (TTL) mechanism in an unstructured DHT 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 overlay network.

Similar to our approach, research such as [9, 12, 21] encodes 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 owning a large fraction of the resource information, especially if there are many resources 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.

Balanced Overlay Networks (BON) [7] encode information about each node's available computational resources,



(a) Job Wait Time (Light Workload)

(b) Job Wait Time (Heavy Workload)

(c) Number of Messages (Light Workload)

(d) Number of Messages (Heavy Workload)

Figure 2: Performance Results with Uniformly Distributed Job Running Times: In the figures the Y-axis does not start from 0%, to show the results more clearly

resulting in a self-organized network that allows jobs to be assigned to free nodes via short random-walks. Similarly, Awan et al. [4] 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. However, the job allocation model in these systems does not consider the resource requirements of the jobs nor the varying resource capabilities of the nodes.

Dynamic load balancing concepts are widely used for distributing loads in locally distributed systems [28] or for thread migration policies [5]. Zhou et al. [33] incorporate dynamic load balancing by employing two distinct scheduling steps: *initial scheduling* and *later migration*. A client initially schedules its jobs on a host in the current *night-time zone* and when the host machine is no longer idle the job is migrated to a new night-time zone. However, they do not allow users to specify resource requirements for the jobs. Therefore, it is a much simpler model than in our design, where a node receiving a migrated job must be able to meet resource constraints of the job. Other researches [11, 34] also investigated dynamic load balancing techniques over structured DHTs, however, they cannot support job constraints during the course of job migrations. The Condor system uses *preemptive resume scheduling*, which can migrate jobs before they complete (*preemption*) in order to meet the needs of system participants (such as owners, users and system administrators) or to deal with the inevitable heterogeneity of available computers [26]. However, Condor is based on a centralized server-client architecture that limits its scalability and robustness to failure, compared to our decentralized P2P desktop grid system.

6. CONCLUSION

In this paper, we have described a P2P desktop grid system that can efficiently match resource requirements for incoming jobs, while simultaneously balancing load among multiple candidate nodes. By introducing lightweight yet effective dynamic load balancing schemes, we have shown experimentally that our system can overcome the load imbalances that arise from the heterogeneity of node capabilities and job running times and also from stale load information.

Desktop grid computing systems such as BOINC [6] or SZTAKI Desktop Grid [29] have still been actively developed and used for supporting various challenging scientific applications. For example, the estimated performance of last 48 hours in SZTAKI was 2046.07GFlop/s with peak performance of 3.4TFlop/s by harnessing 100,000 computing hosts over Internet [29]. We believe that this form of Volunteer Computing [2] will continue to evolve and be a viable choice for supporting most challenging scientific applications consisting of hundreds of thousands of bags of tasks. Our P2P desktop grid system can give an insight into the research community not only by improving the current

centralized desktop grid architectures but also by applying our techniques to wherever autonomic load balancing mechanism becomes crucial to effectively support a very large number of jobs over heterogeneous computing resources.

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd Conference on Computing frontiers*, May 2006.

[6] BOINC: Open-source software for volunteer computing and grid computing. Available at http://boinc.berkeley.edu/.

[7] 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.

[8] 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.

[9] 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.

[10] Folding@Home. Available at http://folding.stanford.edu.

[11] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of the IEEE INFOCOM 2004*, Mar. 2004.

[12] 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.

[13] A. Iosup, C. Dumitrescu, and D. Epema. How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID 2006)*, Sept. 2006.

[14] A. Iosup and D. Epema. Grid Computing Workloads: Bags of Tasks, Workflows, Pilots, and Others. *IEEE Internet Computing*, 15(2):19–26, 2011.

[15] 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 the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)*, June 2007.

[16] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Trade-offs in Matching Jobs and Balancing Load for Distributed Desktop Grids. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, 24(5):415–424, 2008.

[17] J.-S. Kim, B. Nam, M. Marsh, P. Keleher, B. Bhattacharjee, and A. Sussman. Integrating Categorical Resource Types into a P2P Desktop Grid System. In *Proceedings of the 9 IEEE/ACM International Conference on Grid Computing (GRID 2008)*, Sept. 2008.

[18] 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.

[19] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), 1997.

[20] 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.

[21] 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.

[22] I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *Proceedings of the Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS'08)*, Nov. 2008.

[23] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Towards Loosely-Coupled Programming on Petascale Systems. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC'08)*, Nov. 2008.

[24] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC'07)*, Nov. 2007.

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

[26] A. Roy and M. Livny. Condor and Preemptive Resume Scheduling. *Grid Resource Management: State of the Art and Future Trends*, pages 135–144, 2003.

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

[28] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems.

Computer, 25(12):33–44, 1992.

[29] SZTAKI Desktop Grid. Available at http://szdg.lpds.sztaki.hu/szdg/.

[30] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[31] 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.

[32] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner. Creating Personal Adaptive Clusters for Managing Scientific Jobs in a Distributed Computing Environment. In *Proceedings of the Challenges of Large Applications in Distributed Environments (CLADE'06)*, June 2006.

[33] 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.

[34] Y. Zhu and Y. Hu. Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems. *IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 4, 2005*, 16(4):349–361, 2005.