Matching Jobs to Resources in Distributed Desktop Grid Environments*

Jik-Soo Kim, Bobby Bhattacharjee, Peter J. Keleher and Alan Sussman
Department of Computer Science
University of Maryland
College Park, MD 20742
{jiksoo, bobby, keleher, als}@cs.umd.edu

Abstract

Desktop grids use opportunistic sharing to exploit large collections of personal computers and workstations across the Internet and can achieve tremendous computing power with low cost. However, current systems are typically based on a traditional client-server architecture, which has inherent shortcomings with respect to robustness, reliability and scalability. In this paper, we propose a decentralized, robust, highly available, and scalable infrastructure to match incoming jobs to available resources. The key idea behind our proposed system is to leverage information provided by an underlying peer-to-peer system to create a hierarchical Rendezvous Node Tree, which performs the matching efficiently. Our experimental results obtained via simulation show that we can effectively match jobs with varying levels of resource constraints to available nodes and maintain good load balance in a fully decentralized heterogeneous computational environment.

1 Introduction

This paper describes new techniques that employ Peer-to-Peer (P2P) services for robust *desktop grid* computing. Existing platforms for desktop grid computing typically employ a client-server architecture, where a trusted server supplies jobs to a set of client machines distributed across the Internet. Commercial examples of these systems include Entropia [6] and United Devices [22], while non-profit examples are SETI@Home [2], Folding@Home [8] and the BOINC [1] system. In all of these systems, the owner of the server controls which jobs are to be run, the clients request and run jobs when they are able, and the server collects the results. Robustness and reliability come from the server maintaining state about all outstanding jobs being run at potentially unreliable clients, so that jobs assigned to clients can be re-run if a client does not return a result in a time period determined by the computational complexity of the job. The server must therefore reliably maintain state, or the status of outstanding jobs can be lost. The server typically stores the state in a (relational) database, which provides some level of reliability. However, no new jobs can be assigned to a client whenever the server becomes unavailable either due to server failure or network partition.

Our goal is to design and build a massively scalable infrastructure for executing grid applications on a widely distributed set of resources. Such infrastructure must be *decentralized, robust, highly available* and *scalable*, while effectively *mapping* application instances to available resources throughout the system. By employing P2P services, our techniques allow users to submit jobs to be run in the system, and to run jobs submitted by other users on any resources available in the system that meet the minimum job requirements (e.g., memory amount, 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 [15], and a system such as SETI@Home [2] or BOINC [1] for farming out jobs from a server to be run on a potentially very large collection of machines in a completely distributed environment. Such a confluence of P2P and distributed computing is a natural step in the progression of Grid computing, and has indeed been described as inevitable [9, 10, 14]. However, while there has been some research on resource discovery and scheduling for cycle sharing using P2P services [3, 4, 5, 12, 20, 25], no comprehensive set of algorithms and protocols has yet been designed and built, nor a system deployed for matching jobs with different types of resource constraints to a set of widely distributed and heterogeneous resources. Also, as such a system scales to large configurations, matching jobs with different levels of resource requirements to the set of available heterogeneous computational resources becomes a challenging problem.

*This research was supported by the National Science Foundation under Grant #CNS-0509266, and NASA under Grant #NAG5-12652.

In this paper, we describe a set of distributed and decentralized algorithms focusing on submitting jobs and matching them to available resources, and also discuss P2P techniques for both balancing load and for resilience. Our approach uses a novel structure called the *Rendezvous Node Tree*, which allows resource utilization to be efficiently aggregated and disseminated. The Rendezvous Node Tree is built by leveraging routing information from the underlying P2P system.

We quantify the overall behavior of the job scheduling and management algorithms through simulation on a heavily modified version of the *Chord* [21] simulator. Our modifications deal primarily with scheduling and managing jobs, as systems based on *Distributed Hash Tables* (DHTs) generally only provide primitives for object insertion and location. The mechanisms implementing the basic Chord functionality, such as routing requests, inserting objects, and generating events, are largely unchanged.

The rest of the paper is structured as follows. Section 2 is a short introduction to basic peer-to-peer services. Section 3 discusses basic framework services related to managing job placement and monitoring in the decentralized system, while Section 4 provides a comprehensive description of the algorithms and optimization criteria for matching jobs to resources. Section 5 describes our evaluation metrics for the system, and provides simulation results on different types of workloads (sets of jobs). We conclude and discuss future work in Section 6.

2 Related Work: Peer-to-Peer Services

Peer-to-peer research has shown that a robust, reliable system for storing and retrieving files can be built upon unreliable machines and networks. Systems such as Kazaa [13] have been scaled to very large numbers of machines, and support large numbers of simultaneous user requests for files. The algorithms for object location and routing in P2P networks such as CAN [17], Chord [21], Pastry [19], Tapestry [24] and Coral [11] are also capable of scaling to very large number of peers and simultaneous requests for service. These kinds of algorithms are also called *Distributed Hash Tables* (DHTs). Building upon these basic services to provide a system for making computational resources available on demand can allow users to both provide resources when they are not being otherwise used, and to obtain resources when they are needed.

A key distinguishing feature of P2P protocols is *self-organization*. When new peers join, or existing peers leave or fail, the underlying protocols restructure the current state of peers such that overall services are restored without manual interruption. For example, if the service is data storage and lookup, then as peers join or leave the system the underlying protocols replicate and migrate data so that a user is largely unaffected by changes in peer membership. A related property is *scalability*: per peer overhead (in terms of stored state and bandwidth overhead) is often constant or bounded by a logarithmic factor of the number of peers in the system. Such bounds allow these systems to grow to very large number of peers.

Even though our techniques are not specific to any particular DHT algorithms, for the performance characteristics desired we require that peers be assigned IDs chosen uniformly at random from the *Globally Unique ID* (GUID) space (which is universally the case for the DHTs). For the purposes of this paper, however, we assume that the underlying DHT in use is Chord because of its relative simplicity.

We provide a very brief overview of the Chord system here. The service provided by Chord, and indeed any DHT, is that of distributed lookup. DHTs allow objects to be stored and later retrieved whenever they are needed, using their GUIDs. The location at which an object actually resides within the system is determined by the object's GUID, and the GUIDs of the physical nodes (peers) participating in the system. Specifically, Chord's structure may be visualized as a ring with points labeled from 0 to 2^{160} -1. Peers and objects are all assigned GUIDs through an SHA-1 hash [7] of a user-defined name (for objects) or IP address (for nodes). An object is stored at the node on the Chord ring that follows the object's GUID in a clockwise traversal of the ring. The randomization provided by the SHA-1 hash provides a measure of load balance, both in storage and in routing. The structure of the routing tables maintained by each node enables any other node in the system to be reached in $O(\log n)$ hops. Decentralized maintenance algorithms allow the Chord structure to remain connected and stable despite node failures, peers entering and leaving the system, and large fluctuations in lookup load.

3 Basic Framework for Managing Jobs

In this section, we briefly describe our mechanisms for submitting jobs, and managing and monitoring jobs while they are running, including methods for failure detection and recovery.

A *job* in our system is basically the data and associated profile that describes a computation to be performed. A job profile contains several characteristics about the job, such as the client that submitted it, its minimum resource requirements, the location of input data, etc. All jobs in our system are *independent* as described in [16], which means that no communications are needed between jobs. This is a typical scenario in a desktop grid environment, enabling many independent users to submit their jobs to a collection of nodes in the system.

We assume that if a client wants to submit jobs to the system, it can access one of the existing nodes in the sys-

tem, called its *Job Injection Node*, using an externally defined mechanism [21]. The Job Injection Node generates a GUID for the submitted job, using the underlying DHT mechanism, and initiates the *insertion* of the job into the P2P network. Responsibility for the job will be assigned to the node whose GUID is closest in a clockwise direction around the Chord ring to the job's GUID, via the *routing* mechanism. The node that hosts a job is termed the job's *Owner Node* and is responsible for monitoring the execution of the job, whether the job is executed locally or at some other node. The Owner Node is also responsible for ensuring that the job results are returned to the client. Since the Owner Node of a job is determined based on the randomly generated GUIDs, some initial amount of of load balancing is automatically achieved by spreading ownership of the jobs somewhat evenly across the nodes in the system. After successful insertion of a job to an Owner Node, the Owner Node informs the client that the job is ready to run.

When a new job is assigned to an Owner Node, the Owner Node attempts to find an appropriate node for running the job (called *Run Node*) through the *matchmaking* mechanism. Matchmaking is the process of matching jobs with physical resources, and consists of finding an appropriate node for running a job based on the constraints in the job profile and the current (distributed) state of the nodes in the system. The job profile can include several constraints for running the job, such as required CPU speed, amount of memory, supported operating system type, etc. Therefore, in the matchmaking process the first criterion in finding a match is whether the job constraints can be met. After finding one or more nodes that satisfy the job constraints, the matchmaking algorithm can consider balancing load across the nodes. More details about the matchmaking process will be described in Section 4.

Once an appropriate Run Node is found, the new job is inserted into the *job queue* of the Run Node. Each Run Node processes jobs in its job queue in FIFO order and only processes one job at a time. Until a job is completed and its results are returned, the Run Node *periodically* sends a *heartbeat* message to the Owner Node, which can relay the message to the client that initiated the job. This heartbeat message informs the Owner Node about the status of the running job and also indicates that the Run Node is still alive. The Run Node must generate heartbeat messages for every job in its job queue, including jobs that are not yet running. This soft-state heartbeat message plays an important role in failure recovery during the processing of jobs in our system. By employing the Owner Node and Run Node pair, our system can provide a robust environment for processing jobs. Also, the job profile is replicated both on the Owner Node and the Run Node to enable reconstruction of a job in case of failures. If one of the Owner or Run Nodes fails, the other node will detect the failure and recover to make progress in the job execution (we omit the details of the recovery mechanisms). To communicate via the heartbeat message, for efficiency we employ a direct connection between the Run Node and the Owner Node, for example by a socket connection, rather than using the P2P routing mechanism. After completing the job, the Run Node sends the results to the Owner Node of the job. The Owner Node stores the job results and either sends the client the result or sends a pointer to the result (another GUID). The lower part of Figure 1 depicts the overall job lifecycle.

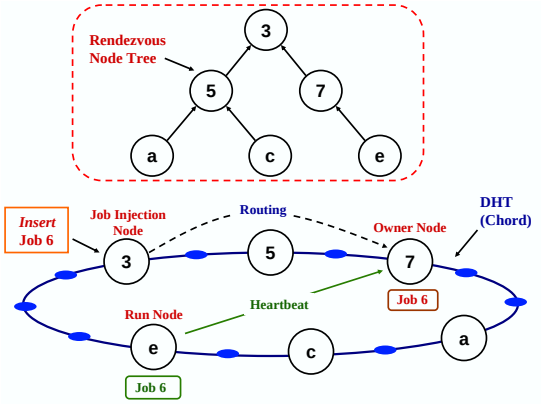


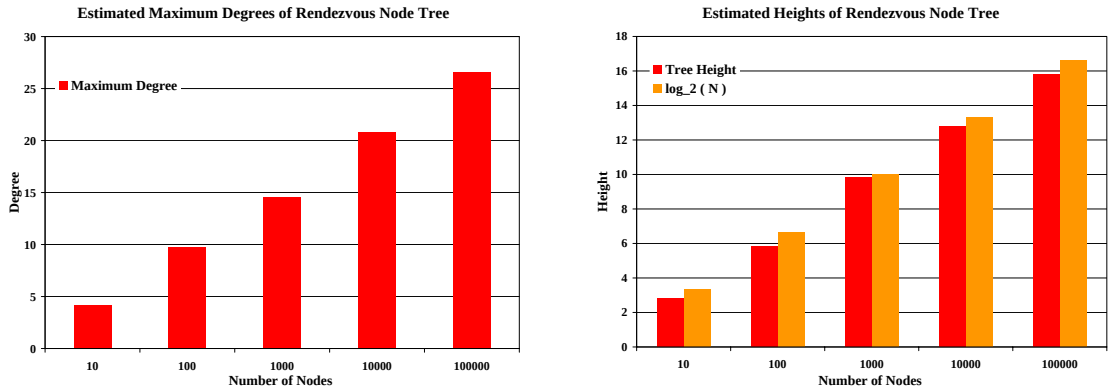
Figure 1. The life cycle of a job in the P2P system (lower), and the corresponding structure of the Rendezvous Node Tree (upper).

4 Matchmaking through the Rendezvous Node Tree

In this section, we present the details of our matchmaking algorithm, given a set of job profiles and a set of node specifications in the P2P system. We employ an overlay network placed on top of the underlying DHT structure, called a *Rendezvous Node Tree* (RN-Tree).

4.1 Building the Rendezvous Node Tree

The RN-Tree is an *implicit* tree built on top of the P2P network, and consists of all currently participating nodes. Each node can determine its RN-Tree parent node based on only local information, which enables building the tree in a completely decentralized manner. The parent-child relationship in the RN-Tree is defined in Equation 1, where *PredID(N)* is the GUID of the *predecessor* of node *N* on the Chord ring, and *Successor(K)* is the node on the ring that is routed to for GUID *K*.



(a) Degree of RN-Tree

(b) Height of RN-Tree

Figure 2. Characterizing the Rendezvous Node Tree

$$Parent(N) = Successor\left(\frac{PredID(N)+1}{2}\right) \quad (1)$$

Equation 1 says that we define the parent node of a node *N* in the tree by mapping to node *Parent(N)* through routing in the P2P network. The Chord system provides both the GUID of the predecessor node in the Chord ring, and the means to find the successor node for any GUID (the default behavior when routing to any GUID in the ring is to route to the node with either that GUID or to the node closest to the GUID in the clockwise direction around the ring). Figure 1 shows a simple example of building an RN-Tree with the set of node GUIDs shown (GUIDs are shown in hexadecimal). This mechanism attempts to build an RN-Tree that is reasonably well balanced and also ensures that *Parent(N)* has a smaller GUID than does *N*. Since the GUIDs of nodes in the system are generated uniformly at random, the overall height of the RN-Tree built based on Equation 1 is likely to be $O(\log N)$ where *N* is the total number of nodes in the entire system. However, even though the GUIDs are generated with uniform distribution through the GUID space, unless the GUIDs for the set of nodes in the system are a perfect arithmetical progression, there can be nodes in the RN-Tree that have more than two children nodes. Thus, the overall RN-Tree may be a *bushy* tree. We simulated creating an RN-Tree up to 100,000 nodes, using the standard C library functions `srand48` and `lrand48` to generate 48-bit GUIDs, and the maximum node degree in the RN-Tree was around 25. Also, the height of the RN-Tree is always close to $\log_2 N$ (see the Figure 2). However, we plan to more formally characterize the structure of the RN-Tree in future work.

Since there can be new nodes joining the system and node departures (leave or fail), the correct parent pointer of a node can change over time. Therefore each node must refresh/update its RN-Tree parent node pointer *periodically* to maintain the RN-Tree appropriately. This mechanism is similar to the *stabilization* mechanism of Chord (and other DHTs) that is used to maintain routing information in the P2P network, and is performed at the same time. The *root* node of the RN-Tree is easy to determine in a Chord ring. If the interval [Predecessor ID + 1, Node ID] includes 0, then the node is the root of the RN-Tree. It is also possible that at some points in time a child node cannot determine its correct parent, because of stale information in the DHT (i.e. the node predecessor and successor information has not yet been updated). However, since the underlying Chord system periodically stabilizes the network, the RN-Tree will be updated too.

4.2 Disseminating Aggregated Resource Information

Once a node finds its RN-Tree parent node, it *periodically* sends local *subtree* resource information (for the subtree rooted by that node) to its parent node, and this information is *aggregated* at each level of the RN-Tree. Therefore, at each level of RN-Tree the aggregated resource information provides an overall picture of the state of the entire subtree. The root of the RN-Tree obtains aggregated resource information about all nodes in the system. To maintain scalability, local subtree information is aggregated at each node in the tree, so the amount of resource information maintained at each node (and communicated from child to parent node) remains *constant*. This notion of *hierarchical*

aggregation is a fundamental abstraction for scalability in a large system and is also used in distributed information management systems [18, 23].

When a parent node receives information from its children nodes, it updates its local child node information and merges all its children information into its aggregated subtree resource information. As in the case of the heartbeat messaging mechanism for monitoring jobs, we employ direct communication between the RN-Tree child and parent nodes for performance reasons, falling back on the P2P routing mechanisms if the direct communication fails.

The most important aggregated resource information that is used in our matchmaking process is the **Aggregated Maximum Available Resources**, representing the maximum amount of each resource available in *some* node in the subtree. The resources modeled include continuous variables, such as the speed of the CPU, the amount of memory available, and the amount of disk space available, and discrete variables such as operating system type and version. The resources modeled match the constraints (requirements) that can be specified in job profiles.

Additional information can easily be added to the aggregated resource information stored and propagated in the RN-Tree. By using the aggregated resource information, each node can determine whether a node with a desired set of resources may exist in its subtree, which is exactly what is needed for the matchmaking algorithms.

4.3 RN-Tree Matchmaking

Based on the disseminated aggregated resource information in the entire RN-Tree, matchmaking for a given job starts from its Owner Node. Each job profile specifies resource constraints that must *all* be satisfied for a node to run the job. A job profile can specify a wild card for a *don't care* condition for a resource. For example, suppose job profiles can specify three different resource requirements for a job: the speed of the CPU, the required amount of memory and the required disk space. Then, each job can specify its constraints for the resources with a tuple [*MinCPU, MinMemory, MinDisk*]. If a job specifies 0 for any resource, it means that the job does not have a specific minimum requirement for that resource in its execution environment.

Algorithm 1 shows the basic matchmaking steps in a node. Each node actively participates in the matchmaking process for jobs, and either processes matchmaking requests locally or forwards them to other nodes. During the search, if a node determines that it can satisfy the job profile constraints, then the matchmaking is done, and the node becomes the Run Node for the job (Step 1). Otherwise, for a given job profile a node can effectively prune the request (i.e. terminate its search) if the resource constraints cannot be satisfied in its entire subtree, based on the aggregated resource information it has available locally. In this case, the node forwards the matchmaking request to its parent node (Step 3). If the node is the root of the RN-tree, so has no parent, then the search fails and the client that submitted the job is notified that no node is available that meets all the constraints. If the request *may* be satisfied in the subtree based on the node's aggregated resource information, then the node searches its children nodes to determine which subtree may contain a node that is able to meet the constraints of the job (and has not been visited yet) and forwards the request to a child node, searching all children nodes if necessary (Step 2). If no child node can satisfy the job profile, the request is forwarded to the parent node (Step 3).

Algorithm 1
Matchmaking in a Node *n* for a given Job *j*

1: **Check Self**: IF *n*.spec meets *j*.constraints RETURN *n* as the Run Node.
2: **Search Children and Forward**: IF Step 1 fails, FIND a child node *c* of *n* where *c*.aggr-spec meets *j*.constraints and *c* has not been visited yet. FORWARD *j* to *c*.
3: **Forward to the Parent**: IF Step 2 fails for all children of *n*, FORWARD *j* to Parent Node *p* of *n*. INFORM *p* that *n* (and *n*'s subtree) has already been visited.

Searching for a node that meets all the constraints of a job can fail, even though the aggregated resource information says that there may be a node in a subtree that meets the constraints. In this case, the matchmaking process *backtracks*, and continues to try to find another possible candidate Run Node in other children nodes in the RN-tree.

We note that there are other factors that can affect the matchmaking process. If the network is very unstable (with many new node joins or node departures), the RN-Tree also becomes unstable and this results in having outdated aggregated resource information or invalid parent node pointers. In such situations, some nodes may not even be able to find their correct RN-Tree parent nodes, so that matchmaking requests cannot be forwarded to higher level nodes in the RN-Tree. However, as soon as the underlying DHT mechanisms stabilize the overall P2P network, the RN-Tree's periodic mechanism for updating aggregated resource information will enable the matchmaking process to make progress.

4.4 Extending the RN-Tree Search

One useful property of the RN-Tree matchmaking algorithm is that if there is at least one node that meets the constraints of a new job, then the matchmaking algorithm *always* finds the node. If the search reaches the root of the RN-Tree and does not find a node that meets the job

constraints, that means that at that time there is no node in the entire system that can run the job. The *basic* RN-Tree matchmaking mechanism ends the matchmaking process as soon as it finds a Run Node that meets the constraints of a job. However, this may lead to poor load balance across nodes for running jobs, since there can be hot spots in the RN-Tree where many jobs are mapped into a comparatively small number of Owner Nodes. Therefore, we *extend* the basic matchmaking process to find *more than one* candidate Run Node and select the final Run Node from among the candidates. One criterion for deciding the *best* Run Node among the candidates is the *size of the job queue* (the current set of unfinished jobs assigned to a node) at the time of the matchmaking. Queue size can either be modeled as the number of jobs in the queue (which was used in our experiments), or an estimate of the run time for all current jobs in the queue. This criterion is similar to the *Minimum Completion Time* (MCT) heuristic in the literature [16] which assigns each job to the machine that results in the earliest completion time of the job.

The extended search algorithm should result in better load balancing across nodes, and this will be shown experimentally in Section 5. The extended search algorithm is different from other matchmaking algorithms described in the literature, which often employ a *Time-To-Live* (TTL) constraint that limits the number of trials (often network hops) to find a resource that meets the constraints of a job [3, 4, 12]. TTL-based matchmaking mechanisms may often fail to find appropriate nodes for running jobs (that meet the job constraints), even though candidates exist somewhere in the network. The extended search mechanism does create additional overhead so increases the matchmaking cost, and this effect is also measured experimentally in Section 5.

5 Evaluation

In this section we describe several *metrics* for evaluating the overall performance of our matchmaking framework for various synthetically generated traffic workloads, and present simulation results. Since we are simulating the behavior of the P2P system, we estimate the performance in terms of the time units employed by the simulator (Chord) and the number of message hops required to perform a matchmaking operation. We tested three different matchmaking algorithms, *Basic RN-Tree Matchmaking* (**RNT-BS**), *Extended RN-Tree Matchmaking* (**RNT-ES**) and *Centralized Matchmaking* (**Central**) for our comparative analysis. More details about the configuration of the matchmaking algorithms will be given in Section 5.2.

5.1 Evaluation Metrics and Traffic Workloads

To measure matchmaking performance, we use two different metrics: *Job Wait Time* (JWT) and *Matchmaking Cost* (MC). JWT measures the time period from the submission of the job until the job gets run on the Run Node, and is a useful metric for evaluating client response time (but doesn't include the time to run the job). MC is the number of messages sent between nodes that is required to find a Run Node for a job, and measures the overhead of performing the matchmaking process using the RN-Tree.

To generate the traffic workloads, we used the *TrafficGenerator* from the Chord simulator, with substantial modifications to produce heterogeneous job profiles and node profiles. A traffic workload consists of three parts: the job profiles, the node profiles and an event arrival pattern. The job profiles specify different levels of resource requirements and amount of computation the job requires (in time units), and the node profiles specify the resource capabilities of each node. The event arrival pattern, produced by the Chord TrafficGenerator, determines when nodes join the system, leave the system, or fail, and also determines when each job is submitted by a client. By varying the event arrival pattern, we can generate widely varying system dynamics and workloads, ranging from highly dynamic, heavily loaded systems to relatively stable systems with lighter overall workloads.

In our experiments, we employed constraints for three different resource types (CPU speed, memory space, and disk space), both for submitted jobs and for node profiles. To generate the node profiles, we used 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. A job profile can specify up to three different resource constraints, denoted by the tuple [MinCPU, MinMemory, MinDisk], specifying the minimum required CPU speed, amount of memory and disk space required by the job. These resource constraints should all be satisfied for a node to run the job. However, some jobs may not want to specify constraints on all resources, so there is some heterogeneity in the number of constraints in the job profiles. To model these kinds of job profiles, we introduce a *Constraint Heterogeneity Factor* (CHF), which enables us to generate various types of job profiles that have different numbers of resource constraints. With a smaller CHF, the proportion of jobs that have fewer constraints (or none at all) increases, while a higher CHF implies that the fraction of jobs with many constraints is greater. Except for the don't care constraints (which are set to a 0 value for that resource as described in Section 4.3), each constraint is set from the cor-

responding resource capability of a randomly selected node from the Present Nodes (see the Table 1) in the system. Therefore, during the simulation, there should always be at least one node in the system that meets the constraints of a job. After generating constraints for the job profiles, the TrafficGenerator also generates the amount of computation required to complete each job, which is also related to the job constraints. The amount of work *W* for a job *j* is generated uniformly at random from a predefined set of work ranges, 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.

A traffic workload consists of the node profiles and job profiles, and models a P2P desktop grid environment with a heterogeneous set of nodes and jobs with different classes of resource requirements and running times. After generating a traffic workload, we characterize the workload using two metrics, *Composition of Job Profiles* (CJP) and *Difficulty of Job Profiles* (DJP). CJP shows the percentage of jobs that have different numbers of constraints, from 0 to 3. For example, the percentage of jobs with only one constraint (CPU, memory, or disk) is shown in the "% of Jobs with 1 Constraint" line in Table 1. The DJP metric shows how many nodes in the system are able to meet the constraints of the jobs at the time matchmaking is performed for those jobs (averaged over all jobs with that number of constraints). We collected these statistics through our *Centralized Matchmaker* algorithm, which has a global view of the entire network at the time matchmaking is performed (more details about the Centralized Matchmaker are described in Section 5.2). For example, the average percentage of nodes that could possibly match the jobs with two constraints at the time they were submitted for matchmaking is shown in the "Difficulty of Jobs with 2 Constraints" line in Table 1.

5.2 Matchmaking Algorithms

One difficulty in evaluating the experimental results for matchmaking is what to use as a good base algorithm for doing the matchmaking, to compare against the RN-Tree matchmaking algorithms. For our comparison, we have designed an *online* scheduling mechanism, called the *Centralized Matchmaker* (Central) that has global information about the current capabilities and load information for all the nodes in the system, so can assign a job to the node that both satisfies the job constraints and has the minimum job queue size across all nodes in the entire system (breaking ties arbitrarily). We model the job queue size as the number of jobs in the queue, as described in Section 4.4. The Centralized Matchmaker is an online algorithm because it makes a decision to assign a job based on the state of the entire system at the time the job is submitted, rather than computing an *offline* optimized assignment from complete *a priori* information about the arrival patterns of jobs and node joins and departures. 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. In our simulation environment, the Centralized Matchmaker does not incur any cost for performing the matchmaking, since we only model the messaging behavior in the P2P network for traversing the RN-tree. We have also tested a naive matchmaking algorithm that does not use any resource information to assign jobs to nodes, called the *Random Matchmaker*. The Random Matchmaker generates a random GUID and uses that as the Run Node for a job, as is done for finding the Owner Node of a job. Note that this does not take into account satisfying the job constraints. However, in experiments not shown, we verified that even with jobs that have no constraints, the Random Matchmaker shows much worse load balance and Job Wait Time compared to the basic RN-Tree algorithm.

We employed two versions of the RN-Tree matchmaking algorithms. The first version, RNT-BS, implements the basic algorithm and the second version, RNT-ES, employs the extended search algorithm with the number of candidate Run Nodes set to 2 (as described in Section 4.4). This means that RNT-ES attempts to find two candidate Run Nodes, and selects the one that currently has a shorter job queue.

5.3 Comparative Analysis

Table 1 shows the characteristics of our test traffic workloads. We tested six different traffic workloads, three of them for relatively stable network environments (S-TR-H40, S-TR-H60, S-TR-H80), and three for more dynamic environments (D-TR-I, D-TR-II, D-TR-III). Our test traffic workloads are generated via the following steps. First, 1000 nodes join the system with an average inter-arrival time of 100 time units. After a stabilization period to allow the Chord network to settle, the next 10000 events (for static workloads) or 15000 events (for dynamic workloads) arrive with an average inter-arrival time of 100 time units. Events can be new node joins, existing node departures (leave or fail) or job submissions into the system. For the static workloads, all 10000 events are job submissions, since those workloads model relatively stable network environments. However, for the three dynamic workloads the 15000 events are composed of all three types of events, generated with different probability distributions for the event types for each workload. All of these generated events are based on the *Poisson Distribution* with an arrival rate of (1 / average inter-arrival time). Since the traffic workloads are

	S-TR-H40	*S-TR-H60*	*S-TR-H80*	*D-TR-I*	*D-TR-II*	*D-TR-III*
Present Nodes	1000	1000	1000	1825	1335	2500
Failed Nodes	0	0	0	454	591	264
Left Nodes	0	0	0	291	594	470
Jobs	10000	10000	10000	12685	12295	12032
CHF	40	60	80	50	50	50
Average Event Inter-Arrival Time	100	100	100	100	100	100
% Of Jobs with 0 Constraints	20.5%	5.7%	0.6%	11.5%	12.0%	12.3%
% Of Jobs with 1 Constraint	42.8%	27.8%	8.7%	36.7%	36.9%	37.2%
% Of Jobs with 2 Constraints	29.6%	43.7%	37.9%	38.5%	37.6%	37.5%
% Of Jobs with 3 Constraints	7.1%	22.8%	52.7%	13.3%	13.5%	13.1%
Diffi culty Of Jobs with 0 Constraint	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Diffi culty Of Jobs with 1 Constraint	49.4%	48.9%	48.7%	50.0%	50.3%	50.1%
Diffi culty Of Jobs with 2 Constraints	25.0%	25.2%	25.2%	25.8%	25.4%	25.5%
Diffi culty Of Jobs with 3 Constraints	12.7%	12.7%	12.7%	13.2%	12.9%	12.9%
Node Departure %	0.0%	0.0%	0.0%	29.0%	47.0%	22.7%

Table 1. Test Traffic Workloads: Present Nodes indicate nodes that are active (still alive) at the end of the simulation. Left Nodes are those that leave the system during the simulation, from a planned departure. Failed Nodes are those that become unreachable without any notice due to node failure or network partition.

generated based on random variables and random functions to build the combination of events, each of the dynamic traffic workload has different numbers of nodes and jobs over time. The Average Event Inter-Arrival Time (=100) was chosen such that during the simulation incoming jobs keep every node in the system busy, to model a relatively heavily loaded environment. In experiments not shown, in lightly loaded environments the RN-Tree basic algorithm (RNT-BS) performs essentially as well as the Centralized Matchmaker with respect to load balance, since free nodes (with job queue size 0) are almost always available.

To measure the effects of job constraint heterogeneity, we increased the CHF from 40 to 80 for the static workloads. As CHF increases, the proportion of jobs that have a small number of constraints decreases, and the proportion of jobs with multiple constraints increases (as seen in the Composition of Job Profiles from Table 1). Also, for jobs with many constraints, matchmaking becomes more difficult because the fraction of nodes that are able to satisfy the constraints is smaller. For example, for static workload S-TR-H40, the average percentage of nodes that can match jobs with three constraints is only 12.7% of all available nodes. For the dynamic workloads, each one has different characteristics with respect to the available nodes over time in the simulated system. In Table 1, Node Departure % shows how many nodes depart (leave or fail) the system during the simulation. For example, for the D-TR-II workload, 47% of nodes depart the system during the simulation, which results in the most dynamic environment among all workloads.

The results in Figure 3 show that employing the extended search algorithm for RN-Tree matchmaking (RNT-ES) has significant performance benefits compared to the basic search algorithm, which helps minimize client response time. For the static workloads, RNT-ES has on average 28% of the Job Wait Time of RNT-BS. For the dynamic workloads, RNT-ES does even better, showing on average only 17% of the Job Wait Time of RNT-BS. It is somewhat surprising that we see larger performance improvements for the more dynamic workloads. Since the RNT-BS algorithm selects the Run Node as the first one found to meet the job constraints, it does not immediately utilize nodes that have recently joined the system as Run Nodes. Also, since for our scenarios the dynamic workloads usually have more nodes available in the system than do the static workloads, RNT-ES more effectively utilizes available resources than RNT-BS. RNT-ES significantly decreases both the Average Job Wait Time and the variance in those times compared to RNT-BS, and even makes a good start toward approaching the load balance characteristics of the Centralized algorithm.

As we increase CHF for the static workloads, we see an increase in Job Wait Time for all three matchmaking algorithms (as seen in Figure 3(a)), since in the higher CHF traffic workloads the majority of jobs have multiple constraints so that only a small fraction of the nodes in the entire system can run these jobs. Because of the difficulty of matching some jobs, the standard deviations for the Job Wait Time are consistently high across all the matchmaking algorithms (as seen in Figures 3(c) and 3(d)). Also,

as the overall P2P network becomes very dynamic and unstable, the RN-Tree also becomes unstable resulting in a substantial overhead for performing RN-tree matchmaking. Indeed, as seen from Figures 3(b) and 3(d), the RN-Tree matchmaking algorithms show the worst performance for the D-TR-II workload (among dynamic workloads), which has the highest node departure percentage.

Matchmaking Cost (MC) measures the overhead of the RN-Tree matchmaking algorithms. Figures 4(a) and 4(b) show that the RNT-ES algorithm does not cause substantial additional overhead compared to RNT-BS, in counting the number of messages required to find an appropriate Run Node from two candidates. We used 75th Percentile of MC across all matches in the graphs, instead of the median value, since for the median the matchmaking cost is sometimes very small because large numbers of jobs have few constraints in some workloads. Across all the workloads, RNT-ES can find an appropriate Run Node that matches the job constraints for the 75% of all jobs that had the lowest matchmaking cost, by searching on average only 0.4% of all the available nodes in the system as seen from Figure 4(a) and 4(b) (in the worst case, RNT-ES searches on average 31.7% of all the available nodes in the system). In addition, RNT-ES finds only one additional candidate Run Node, which does not cause much additional cost over RNT-ES. However, if we want to find more than two candidate nodes for a Run Node to further improve load balance, the matchmaking cost will increase accordingly. We will look at the tradeoff between matchmaking cost and load balance more closely in future work. As seen from Figures 4(a) and 4(c), higher CHF can also affect the overall matchmaking cost since the RNT algorithms must find a Run Node that meets all job constraints among a relatively small fraction of the available nodes. The dynamics of the system can also affect matchmaking performance, since frequent node joins and departures can make the RN-Tree unstable and delay the matchmaking until the RN-Tree (and entire P2P network) is stabilized periodically. Indeed, the high standard deviations for matchmaking cost in Figure 4(c) and Figure 4(d) indicate that some jobs suffer in finding an appropriate Run Node due to their difficult-to-meet constraints or from an unstable network at the time of the matchmaking.

We also compared the overall number of messages to perform matchmaking and maintain the RN-Tree (called *RNT Messages*) against the number of messages required to maintain the underlying P2P network for Chord and manage the submission and execution of jobs (called *Basic Messages*). The RNT Messages include the required messages for performing matchmaking using the RN-Tree (i.e., traversing the RN-Tree), periodically refreshing parent information in the RN-Tree, and periodically disseminating aggregated resource information. Basic Messages are those for maintaining the underlying P2P network (e.g., the Chord periodic stabilization mechanism), submitting jobs and generating heartbeat messages during job executions. Since the Centralized matchmaker does not have any matchmaking cost and does not require RNT Messages, it requires only the Basic Messages for its algorithm to maintain the P2P network. As seen from Figures 5(a) and 5(b), the number of additional messages required by the RN-Tree algorithms is small compared to the number of messages required by the underlying P2P system and basic job management. This is because of the characteristics of the RN-Tree, with height bounded by O(log N), and the effectiveness of the search pruning mechanism using aggregated resource information. One point to note is that the average number of Basic Messages required for the RNT-BS scheme is usually larger than for either RNT-ES or Central. Since the RNT-BS algorithm has relatively poor load balancing for the static and dynamic workloads compared to the RNT-ES and Central algorithms, jobs are likely to remain longer in the job queues on the nodes. This causes more heartbeat messages to be generated, since those are generated for *all* jobs in the job queue, which accounts for the increase in the number of Basic Messages for the RNT-BS algorithm. For the dynamic traffic workloads, since there are many new node joins and departures the RN-Tree matchmaking mechanisms sometimes require retrying the search for appropriate Run Node(s), mainly from instability in the structure of the RN-Tree, resulting in an increase in the number of RNT Messages (Figure 5(b)) compared to the static workloads (Figure 5(a)). However the number of RNT Messages still remains quite small compared to the number of Basic Messages even in dynamic, relatively unstable environments.

We can view the Centralized Matchmaker algorithm as the extreme case of the RN-Tree extended search algorithm, since it first finds *all* candidate Run Nodes that meet the job constraints and picks the one with the shortest job queue. Even though the Centralized Matchmaker does not have any matchmaking cost in our simulations, since the simulator can maintain global information about all the nodes in the system, such a scheme would not be feasible in a complete system implementation, since it would incur a large overhead to find *all* nodes in the P2P system that meet the job constraints.

6 Conclusions and Future Work

Desktop grid systems have been shown to be a low cost way to supply computing power to large scale problems, by enabling opportunistic sharing of distributed computational resources, often across the Internet. To robustly and efficiently execute applications on a widely distributed set of resources, it is desirable for a desktop grid system to be decentralized, robust and highly available. In this paper, we proposed new methods to support these requirements, by

employing P2P services, to enable users to both submit jobs to be run in the P2P system and to run jobs submitted by other users. However, as such a system scales to large configurations, matching jobs with different levels of resource requirements to the set of available heterogeneous computational resources becomes a challenging problem. We described how to build an implicit Rendezvous Node Tree on top of an underlying P2P network to effectively match jobs to the nodes with varying capabilities. We also described basic matchmaking algorithms that use the RN-tree, and extended those algorithms to improve load balance across the nodes in the system. Our experimental results, using a modified version of the Chord simulator, show that we can always find a node that meets the constraints of a job if there is at least one node that meets the constraints in the system, and with low overhead for the matchmaking. Also, by employing the extended search matchmaking algorithm, we can achieve better load balance in the fully decentralized system than with the basic algorithm. We also compared our results against a centralized online algorithm, and the results show that the behavior of the extended search algorithm can approach the behavior of the centralized algorithm, with relatively low additional cost for the matchmaking.

In the near future, we will further investigate the behavior of the extended search matchmaking algorithm, to look at the tradeoff between improved load balance and higher matchmaking cost. Other issues that we will address in future work include security aspects such as authentication to restrict access to a set of users, and result verification in an untrusted computational environment. Finally, since all of our experimental results have come from simulation, we plan to build and deploy to our application area collaborators a prototype system based on these and additional simulation results, to look at issues that will arise in deploying a peer-to-peer desktop grid system in a heterogeneous environment running real applications.

References

[1] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the Fifth International Workshop on Grid Computing*, Nov. 2004.

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

[3] 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 USENIX Virtual Machines Research and Technology Syposium (VM 2004)*, May 2004.

[4] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. In *Proceedings of 2003 ACM/IEEE conference on Supercomputing*, Nov. 2003.

[5] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Netwotk. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(3), May 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, 2003.

[7] FIPS 180-1. Secure Hash Standard. *U.S. Department of Commerce/NIST, National Technical Information Service*, Apr. 1995.

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

[9] I. Foster and R. Grossman. Data Integration in a Bandwidth-rich World. *Communications of ACM*, 46(11):50–57, 2003.

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

[11] M. Freedman, E. Freudenthal, and D. Mazi. Democratizing content publication with Coral. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI 2004)*, Mar. 2004.

[12] A. Iamnitchi and I. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Proceedings of the Second International Workshop on Grid Computing*, Nov. 2001.

[13] Kazaa. Available at http://www.kazaa.com.

[14] J. Ledlie, J. Schneidman, M. Seltzer, and J. Huth. Scooped Again. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[15] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th Internatianal Conference on Distributed Computing Systems*, June 1988.

[16] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freud. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59(2), Nov. 1999.

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

[18] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2), May 2003.

[19] A. Rowstran and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[20] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. In *Proceedings of Fifth International Workshop on Global and Peer-to-Peer Computing*, May 2005.

[21] 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 ACM SIGCOMM*, Aug. 2001.

[22] UnitedDevices. Available at http://www.ud.com.

[23] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of ACM SIGCOMM*, Aug. 2004.

[24] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), Jan. 2004.

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

A Generating Traffic Workloads

To generate the node profiles, we used the notions of *Spec Basis* and *Base Resource*. The Spec Basis ranges from 1 to *N* (total number of Spec Basis), so that all nodes in the system are clustered into *N* groups. The larger Spec Basis group consists of nodes that have rare and larger resource capabilities and the smaller Spec Basis group is composed of nodes that have smaller available resource capabilities. To simplify the generation of node profiles, each node that belongs to a group with Spec Basis *B* has a resource with size is around *B* * BASE_RESOURCE (modeling some *jitter* in the size of the resource). The BASE_RESOURCE is the minimum size of that resource in the system, such as CPU speed, amount of memory or disk space. Note that the Spec Basis is defined in terms of each resource independently. For example, a node can have a CPU that belongs to Spec Basis group 1 for its CPU, a memory that belongs to Spec Basis 4 for memory, and a disk that belongs to Spec Basis 2 for disk. In this case, the node has BASE_CPU speed of CPU, 4*BASE_MEMORY of main memory and 2*BASE_ DISK of disk space. The probability function for the Spec Basis grouping is defined by the following formulas:

$$T = \frac{1}{1} * C + \frac{1}{2} * C + \ldots + \frac{1}{N} * C \quad (2)$$

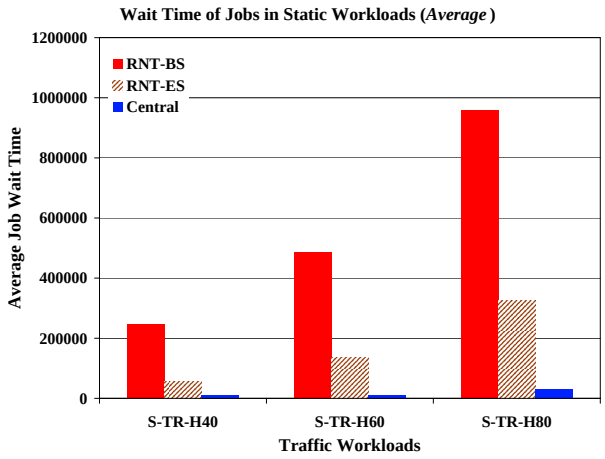
$$Prob\ (Spec\ Basis\ B\ group) = \frac{\frac{1}{B} * C}{T} \quad (3)$$

In the above formulas, *C* is a constant. Therefore, by using this probability function, the distribution of each resource specification is according to function *f(x) = 1 / x*.

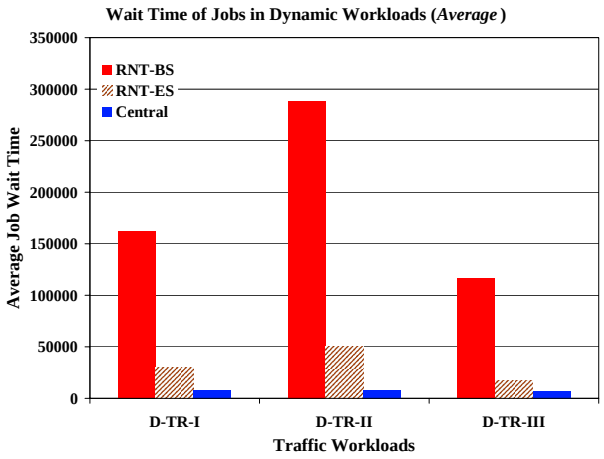
The *Constraint Heterogeneity Factor in job constraints* (CHF) enables us to generate various kinds of job profiles that have different degrees of resource constraints. The CHF can range from 0 to MAX_CHF. CHF of 0 implies homogeneous job profiles, where all of the jobs do not have any resource constraints. In contrast, MAX_CHF means fully heterogeneous job profiles where all of the jobs specify three resource constraints, i.e., every job has minimum speed of CPU, amount of memory and disk space requirements. The CHF can be processed as an input parameter in our TrafficGenerator to generate different kinds of job profiles. Given a CHF, a job profile is generated through the following steps.

1. Pick up a random node *N*.
2. For each resource *R* constraint, generate a determinant *D* uniformly at random in the range [0, MAX_CHF].
3. If $D \leq$ CHF, then Constraint(*R*) = R of node *N*. Otherwise, 0.

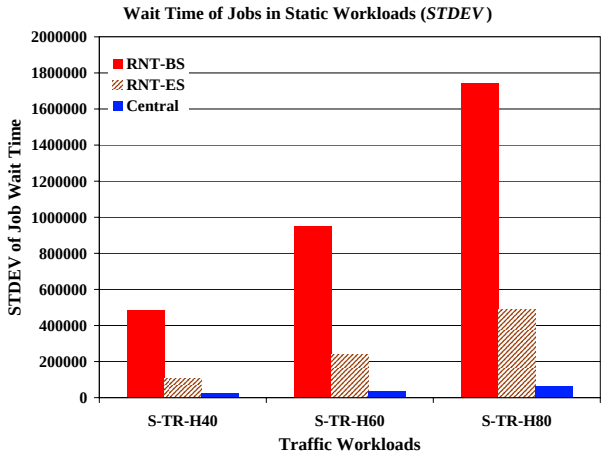
If Constraint(*R*) is set as 0, the job does not have any constraint on resource *R*, i.e. a don't care condition. For example, suppose the generated constraints of a job profile are [100, 500, 0] which corresponds to a [MinCPU, MinMemory, MinDisk] tuple. Then, this job requires 100 speed of CPU and 500 memory unit, but does not have a minimum disk space requirement. With higher CHF, each constraint of a job can be a non-zero value (which comes from the actual node's resource specification) with higher probability, resulting in having more resource constraints to run the job.



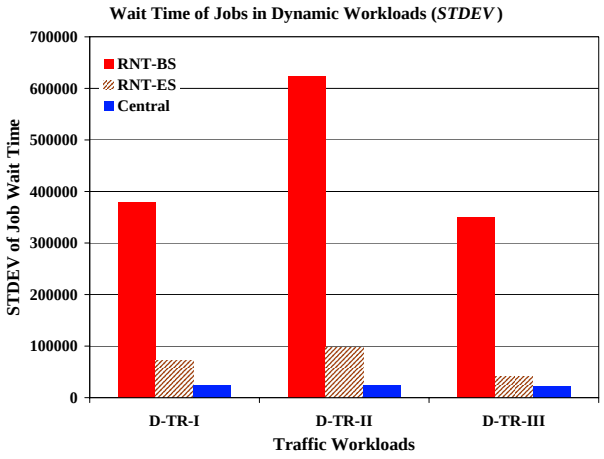
(a) Average for Static Workloads



(b) Average for Dynamic Workloads

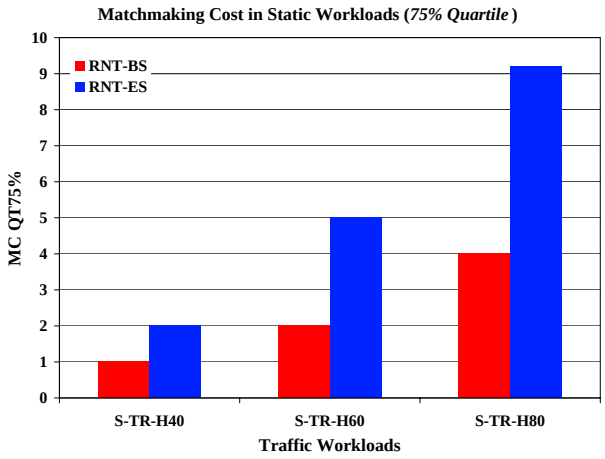


(c) STDEV for Static Workloads

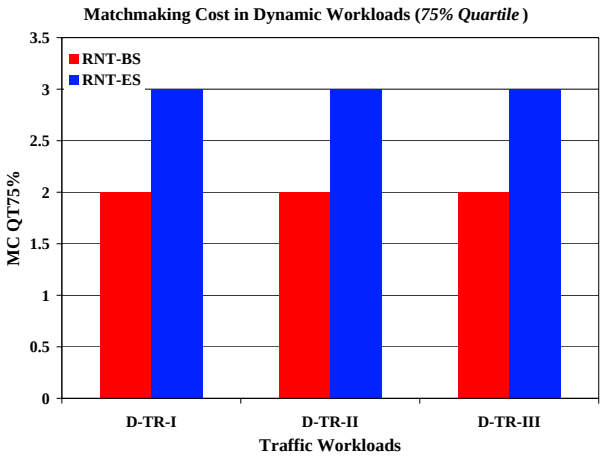


(d) STDEV for Dynamic Workloads

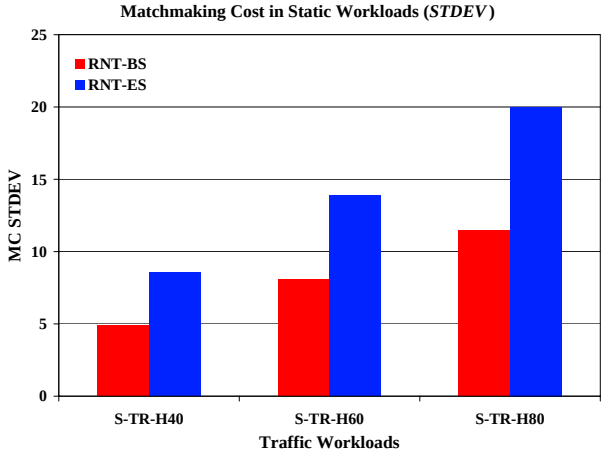
Figure 3. Job Wait Time for Static and Dynamic Workloads



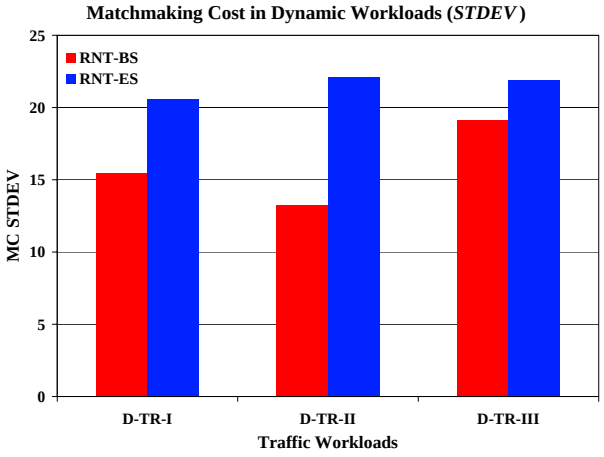
(a) 75th Percentile for Static Workloads



(b) 75th Percentile for Dynamic Workloads

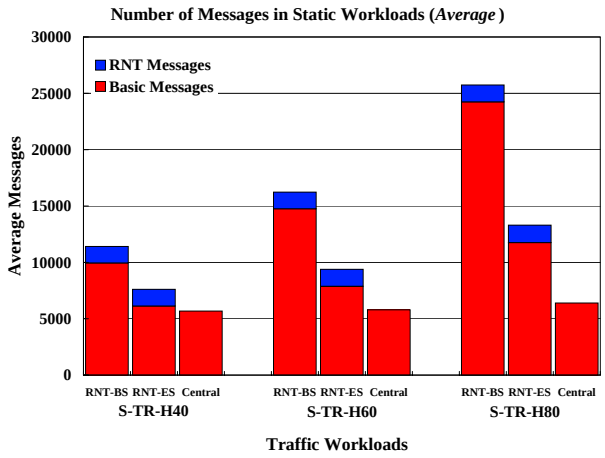


(c) STDEV for Static Workloads

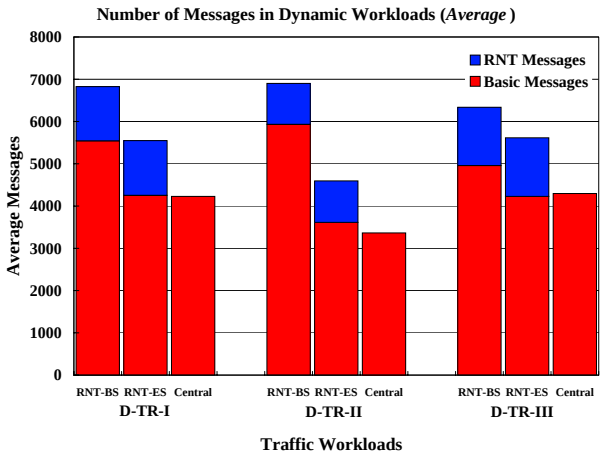


(d) STDEV for Dynamic Workloads

Figure 4. Matchmaking Cost for Static and Dynamic Workloads



(a) Average Messages for Static Workloads



(b) Average Messages for Dynamic Workloads

Figure 5. Number of Messages Sent Per Node in Static and Dynamic Workloads.