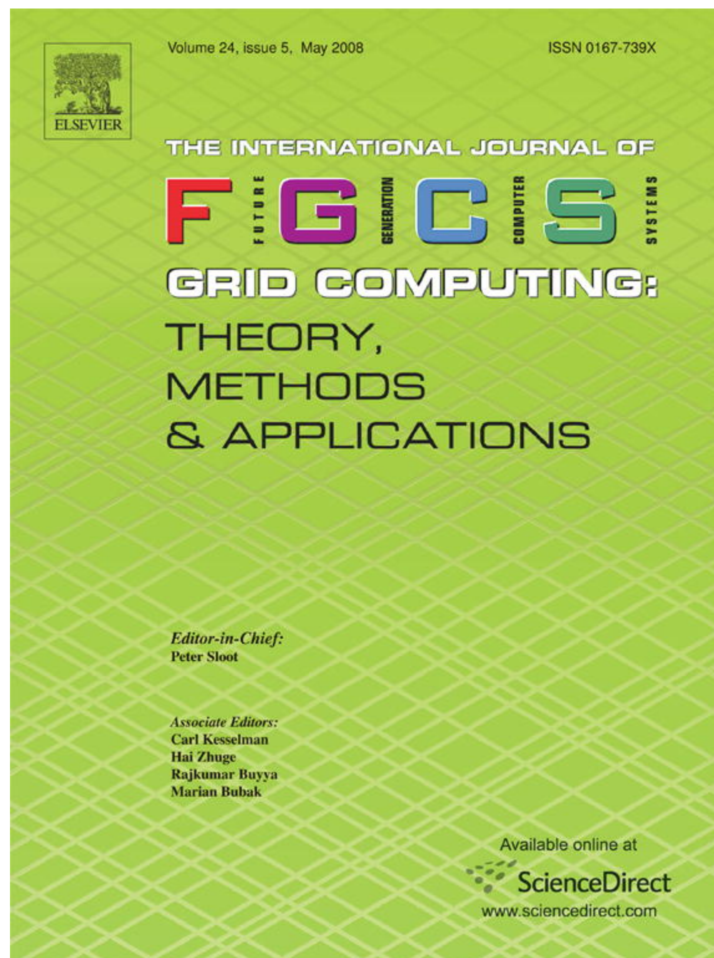
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Available online at www.sciencedirect.com

ScienceDirect

Future Generation Computer Systems 24 (2008) 415–424

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/fgcs

Trade-offs in matching jobs and balancing load for distributed desktop grids☆

Jik-Soo Kim*, Beomseok Nam, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, Alan Sussman

UMIACS, Department of Computer Science, University of Maryland, United States

Received 27 February 2007; received in revised form 12 May 2007; accepted 14 July 2007
Available online 26 July 2007

Abstract

Desktop grids can achieve tremendous computing power at low cost through opportunistic sharing of resources. However, traditional client–server Grid architectures do not deal with all types of failures, and do not always cope well with very dynamic environments. This paper describes the design of a desktop grid implemented over a modified Peer-to-Peer (P2P) architecture. The underlying P2P system is decentralized and inherently adaptable, giving the Grid robustness, scalability, and the ability to cope with dynamic environments, while still efficiently mapping application instances to available resources throughout the system.

We use simulation to compare three different types of matching algorithms under differing workloads. Overall, the P2P approach produces significantly lower wait times than prior approaches, while adapting efficiently to the dynamic environment.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Desktop grid; Peer-to-Peer computing; Resource discovery; Matchmaking; Load balancing

1. Introduction

The recent growth of both the Internet and the hardware capabilities of personal computers and workstations enables distributed computing to achieve tremendous computing power by harnessing tens of thousands to millions of machines. These systems are often called *desktop grid* computing systems and leverage unused capacity on high-performance desktop PCs [1–3]. Desktop grid computing systems mainly target complex scientific applications requiring massive computing power and resources that might exceed those available in a single supercomputing platform. However, existing platforms for desktop grid computing typically employ a client–server architecture, which has inherent shortcomings with respect to robustness, reliability and scalability since the server can be a single point of contention and failure.

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 (called *matchmaking*) throughout the system. Fortunately, these are precisely the characteristics promised by new techniques and approaches in Peer-to-Peer (P2P) systems. Using P2P services can provide a robust, reliable, and scalable job submission and execution system that is able to efficiently utilize widely distributed available computational resources. By employing P2P services, our system allows 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, essentially allowing a group of users to form an ad hoc set of shared resources. Such a confluence of P2P and distributed computing is a natural step in the progression of Grid computing, and has indeed been described as inevitable [4,5]. However, as such a system scales to large configurations and heavy workloads it becomes a challenging problem to efficiently match jobs with different resource requirements to available heterogeneous computational resources, to provide good load balancing, and to obtain high system throughput and low job turnaround times.

In this paper, we extend our previous work [6] and analyze quantitatively the trade-offs between performing efficient matchmaking and achieving good load balance. Via a simulation study, we perform a comparative analysis of three

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

* Corresponding author.
E-mail addresses: jiksoo@cs.umd.edu (J.-S. Kim), bsnam@cs.umd.edu (B. Nam), keleher@cs.umd.edu (P. Keleher), mmarsh@cs.umd.edu (M. Marsh), bobby@cs.umd.edu (B. Bhattacharjee), als@cs.umd.edu (A. Sussman).

0167-739X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.07.007

Author's personal copy

different matchmaking algorithms for several different types of workloads. This study is intended to give insight into the design and implementation of resource discovery algorithms in a distributed and heterogeneous Grid environment. The rest of the paper is structured as follows. Section 2 discusses our assumed context and overall goals. Section 3 describes the algorithms and optimization criteria for matching jobs to resources, while Section 4 contains our evaluation. Finally, Section 5 presents related work and Section 6 concludes the paper.

2. Workload assumptions and overall goals

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

Our expected environment and usage make this problem easier in some ways and more difficult in others. A large fraction of nodes in the system might belong to one of a small number of equivalence classes in terms of their resource capabilities. For example, many organizations buy clusters of identical machines all at once, to create compute farms or just to replace an entire department's machines. Node clusters make the problem more difficult by removing the notion of a single best match for a given job, since the overall system can be composed of sets of homogeneous clusters of nodes. The underlying matchmaking algorithm must be able to cope with many similar nodes and perform some intelligent load balancing across them. However, node clustering can also simplify the problem by reducing the set of possible choices for the matchmaking algorithm. Similarly, job profiles might show clustering in terms of their minimum resource requirements. Sets of similar jobs can result from running the same application code with slightly different parameters or input datasets. For example, researchers often perform parameter sweeps to optimize algorithmic settings or explore the behavior of physical systems. Similarly, the same computation may be performed on different input regions, such as n-body or weather calculations that differ only in spatial coordinates.

Therefore, the overall problem space for Grid computing environments can be divided along two axes, measuring the degree to which the nodes and jobs are either *clustered* or *mixed* (heterogeneous). Systems such as Condor [7,8] mainly target mixed jobs (in terms of minimum resource requirements) in clustered nodes (in terms of resource capabilities), while systems like BOINC [3] or SETI@Home [2] mainly deal with clustered jobs (where a cluster is essentially equivalent to a BOINC project) in mixed nodes. Our intent is to effectively support all the four scenarios. To summarize, the goals of any matchmaking algorithm must include the following:

(1) *Capability*—The matchmaking framework should be able to allow users to specify minimum requirements for any type of resource (CPU speed, minimum memory size, etc.).
(2) *Load balance*—Load (jobs) must be distributed across the nodes capable of executing them.
(3) *Precision*—Resources should not be wasted. All other issues being equivalent, a job should not be assigned to a node that is over-provisioned with respect to that job.
(4) *Completeness*—A valid assignment of a job to a node must be found if such an assignment exists.
(5) *Low overhead*—The matchmaking must not add significant overhead to the cost of executing a job. This may be challenging, given that the matchmaking is done in a completely decentralized fashion.

There are additional issues that are outside the scope of this paper. For example, in some situations (e.g., conditions of low load), the system might prefer to optimize throughput by executing jobs on the *most* capable available node. This raises the question of what we wish to optimize for: throughput or response time. We are explicitly avoiding this issue by designing infrastructure that can accommodate either objective. There are also various security issues related to who can contribute nodes or submit jobs within a single grid system that are not discussed in this paper.

3. Matchmaking algorithms

We begin by defining the terminology and the basic framework of our approach to matchmaking, and then describe the two approaches that we evaluate in this paper: a *Content-Addressable Network* and a *Rendezvous Node Tree*-based mechanism.

3.1. Basic framework

All aspects of the system design assume an underlying *Distributed Hash Table* (DHT) infrastructure [9,10]. DHTs use computationally secure hashes to map arbitrary identifiers to random nodes in a system. This randomized mapping allows DHTs to present a simple insertion and lookup API that is highly robust, scalable, and efficient. We insert both nodes and jobs into a single DHT, performing matchmaking by mapping a job to a node via the insertion process, and then relying on that node to find candidates that are able and willing to execute the job. By leveraging such an architecture, we are effectively *reformulating* the problem of matchmaking to one of routing in the P2P network.

A *job* in our system is the data and associated profile that describes a computation to be performed. A job profile contains all characteristics of the job, including the client that submitted it, its minimum resource requirements, the location of its input data, etc. 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. All jobs in the system are *independent*, which implies that no communication is needed between them. This is a typical scenario in a desktop grid environment, enabling many independent users to submit their jobs to a collection of node resources in the system.

Clients insert jobs into the system by submitting them to any system node. Nodes receiving submitted jobs assign them *Globally Unique IDentifiers* (GUIDs) by using the underlying

Author's personal copy

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

Run nodes periodically send *heartbeat* messages to the owner nodes of all jobs either running or queued locally. Heartbeats are communicated directly between run nodes and owner nodes, rather than through DHT routing. This soft-state message plays an important role in failure recovery during the processing of jobs in our system, as job profiles are replicated on both the owner and run nodes. If either the owner node or the run node fails, the other will detect the failure and initiate a recovery protocol so that the job can continue to make progress. If both fail before the recovery protocol completes, the client must resubmit the job. After a job completes, the run node returns the results to the owner node, which forwards them to the client. More details about our basic framework for job submission and execution in the P2P network can be found in Kim et al. [11].

3.2. Content-Addressable Network

A Content-Addressable Network (CAN) is a DHT that maps GUIDs to points in a d-dimensional space [9] so that the nodes divide up the CAN space into rectangular *zones* and each node maintains *neighbor* information. The conventional use of CAN is to map a GUID into the space by applying d different hashes, one for each dimension. However, positions in the CAN space need not be created through randomized hashes. For example, Tang et al. [12] map documents and queries into a CAN space where each dimension measures the relevance of a particular index term, executing queries via a blind local search centered on a query's mapping.

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

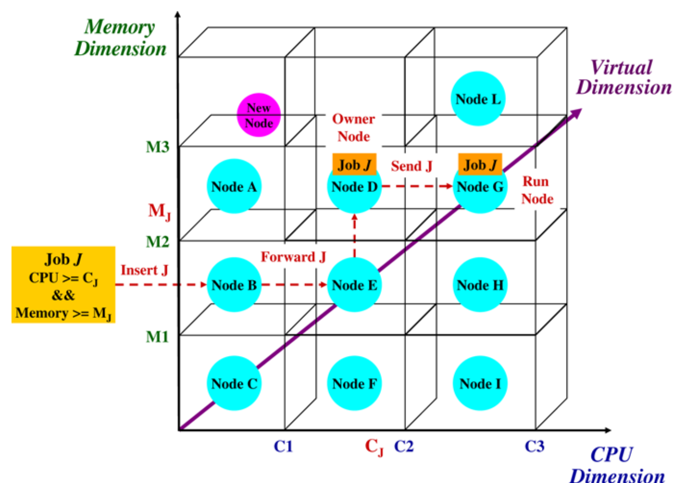


Fig. 1. Matchmaking mechanism in CAN.

meeting a job's constraints can run the job, but a node whose coordinate in any dimension is less than that specified by the job's constraints, even if very close in the CAN space, is not a viable choice to run the job. Hence, instead of searching for the node whose capabilities are closest to the job's constraints, our matchmaking/routing procedure must search for *a node whose coordinates in all dimensions meet or exceed the job's constraints*.

Fig. 1 shows the procedure for matching a job J to the Node G in a system with two resource types, CPU speed and Memory size, through routing in the CAN space. A job is inserted into the system using its requirements as coordinates ($\{C_J, M_J\}$ for Job J) and defining the owner of the resulting zone as the owner node of the job (Node D). The owner node creates a list of candidate run nodes, and chooses the (approximately) *least loaded* among them (Node G) based on load information periodically exchanged between neighboring nodes. To determine the least loaded node among the candidate run nodes, we use the *size of its job queue* (the current set of unfinished jobs assigned to a node) at the time the matchmaking is performed. Queue size can be modeled as either the number of jobs in the queue (which was used in the experiments in this paper) or an estimate of the run time for all current jobs in the queue. Job queue sizes can be included in the *periodic neighbor state update messages* of CAN that are propagated to neighboring nodes [9]. No global synchronization is required, and the additional overhead is a small fixed cost for each update message, sent only to direct neighbors.

By selecting the least loaded node as the *best* run node, we address the problem of *Load balance*, as described in Section 2. The candidate nodes are drawn from the owners of neighboring zones, such that each candidate is at least as capable as the original owner node of a job in all dimensions (capabilities), but more capable in at least one dimension (Nodes G and L). *Precision* and *Capability* follow naturally from the fact that the owner node of a job maintains the zone containing the representative point of a job (corresponding to its minimum resource requirements), so the minimally capable nodes for a job are neighbors (or next-nearest neighbors) of the owner node.

Author's personal copy

Also, under the assumption that there is always at least one node capable of running a given job, *Completeness* is assured by the CAN routing, which in the worst case will eventually map a job to the most capable node in the system (the node occupying the extreme corner of the CAN space). In this special case, the node to which the job is mapped by CAN routing will have to become the run node and select a neighbor to act as the owner node.

The above procedure works in all cases, but may cause some problems for the CAN mechanisms when many nodes have similar, or perhaps identical, resource capabilities. Since the coordinates of a node are defined by its resource capabilities, identical nodes are mapped to the same place in the CAN volume (New Node and Node A in Fig. 1). The best way to distribute ownership of a zone across multiple such nodes is not immediately obvious. Conversely, many jobs might have very similar requirements. For example, many jobs will likely be inserted into the system with no requirements specified at all. In this case, all those jobs will be mapped to the single node that owns the zone containing the minimum point in the CAN volume (Node C).

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

3.2.1. Changes to original CAN

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

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

The second major change to the CAN algorithms is in how zones are merged. A zone is merged with a neighbor when it is orphaned because of an owner leaving, either gracefully or by failure. The default CAN recovery algorithms allow such an orphaned zone to be merged with any neighboring zone: no restriction is made on which nodes can own a zone. In fact, a node can own multiple zones, which can result in a highly fragmented coordinate space. Therefore, to achieve a one-to-one node to zone assignment, CAN runs a periodic *background zone reassignment* algorithm. That algorithm can assign one of the neighbor nodes of the departed node to another region, without any restrictions on merging and reassigning the orphaned zone (for details see Ratnasamy et al. [9]). However, in our system this can violate the required semantics about the relationship between a zone and the owner of that zone, whereby a zone should contain the coordinates (i.e., resource capabilities) of its owner.

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

3.3. The Rendezvous Node Tree

The *Rendezvous Node Tree* (RNT) is a distributed data structure built on top of an underlying DHT, which in our implementation is Chord [10]. Specifically, the RNT copes with the *Load balance* issue by performing a tree traversal after the random initial mapping, and addresses *Completeness* by passing information describing the most capable reachable node up and down the tree.

An RNT contains all participating nodes in the desktop grid. Each node determines its parent node based only on local information, which enables building the tree in a completely decentralized manner (to find the parent node in the RNT, divide the GUID of the *predecessor* node of the child node in the Chord ring by two and find the *successor* node of that

Author's personal copy

GUID in the Chord ring—see details in Kim et al. [13]). Since the GUIDs of nodes in the system are generated uniformly at random, the overall height of the RNT is likely to be $O(\log N)$ where N is the total number of live nodes in the system (we investigated the characteristics of the RNT in terms of overall height and node degree in Kim et al. [13]). Due to the dynamics of the system (new nodes joining, existing nodes departing), the correct parent pointer of a node can change over time. Therefore each node must refresh/update its RNT parent node pointer periodically to maintain the RNT structure.

Once the parent–child relationship in the RNT is determined, each node periodically sends local subtree resource information (for the subtree rooted by that node) to its parent node, and this information is *aggregated* at each level of the RNT (*hierarchical aggregation*). In the work described in this paper, the only information distributed through the tree is a description of the *maximal amount of each resource* available at some node in the subtree.

We inject a job into the system by mapping it to a randomly chosen node, which becomes the job's owner node. This achieves good initial load balancing by spreading the jobs randomly across nodes in the system. The owner node then initiates a search for a run node, which must satisfy the job's resource requirements. The search first proceeds through the subtree rooted at the owner node, only searching up the tree into subtrees rooted at the ancestors of the owner node if the subtree does not contain any satisfactory candidates. The search is pruned using the maximal resource information carried by the RNT. Rather than stopping at the first candidate capable of executing a given job, the search proceeds until at least k capable nodes are found (called *extended search*). The search completes by choosing the least loaded of the k nodes to run the job (as described in Section 3.2). Through experiments not discussed here, we have determined that a value of five (5) for k produces robust results with low overhead. Further details about this search procedure can be found in Kim et al. [13].

3.4. Centralized Matchmaker

To compare against the CAN- and RNT-based matchmaking algorithms, we have designed an *online* scheduling mechanism, called the *Centralized Matchmaker*, that maintains global information about the current capabilities and load information for all the nodes in the system, and so can assign a job to the node that both satisfies the job constraints and has the minimum job queue size across all nodes in the entire system (breaking ties arbitrarily). In our simulation environment, the Centralized Matchmaker does not incur any cost for gathering the global information about the nodes in the system and performing the matchmaking (since the simulator can maintain global information about all the nodes in the system). Even though the matchmaking performed by the Centralized Matchmaker is not always optimal (since it is an online algorithm), it should provide good load balancing and is a good comparison model for other matchmaking algorithms [14,15].

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

4. Performance evaluation

In this section, we evaluate our matchmaking algorithms in decentralized and heterogeneous environments and present a comparative analysis of experimental results obtained via simulations.

4.1. Experimental setup

We use synthetic job and node mixes to simulate the behavior and measure the performance of both the CAN- and RNT-based approaches. Our intent is to model a P2P desktop grid environment with a heterogeneous set of nodes and jobs. We therefore developed an event-driven simulator and generated a variety of workloads, each describing a set of nodes and events. Events include node joins, departures (graceful or otherwise), and job submissions. The events are generated using a Poisson distribution with an arrival rate of $1/\tau$ (τ is the average event inter-arrival time and is set to 0.1 s). Jobs can specify constraints for three different resource types: CPU speed, memory, and disk space. We generated node profiles using a *clustering model* to emulate resources available in a heterogeneous environment, where a high percentage of nodes have relatively small values for their available resources and a small fraction of nodes have larger amounts of available resources [16].

Our first four test workloads are relatively static; no nodes join or leave during the course of the experiments (after 1000 nodes join the system, 10 000 jobs arrive at the system with an arrival rate of τ). The workloads differ on the two axes. Workloads are categorized as either *clustered* or *mixed* (as described in Section 2). The former divides all nodes and jobs into a small number of equivalence classes, where all the items in a given equivalence class are identical. The latter assigns node capabilities and job constraints randomly. Workloads are also distinguished by whether the jobs have *light* or *heavy* constraints. For a given job, each type of resource has a fixed independent probability of being constrained: light jobs have an average of 1.2 constraints (out of the 3) and heavy jobs have an average of 2.4. As a job has more resource requirements (heavy constraints), it is likely to be harder to match the job since fewer nodes in the system can meet those multiple constraints.

The amount of work W for a job j is generated uniformly at random from a predefined set of work ranges (200 s on an average), which 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

Author's personal copy

some baseline node CPU speed), to get a run time on the node a job is assigned to. Finally, for the network communication cost, the latency of a packet between any two nodes in the system is modeled by an exponential distribution with a mean of 50 milliseconds.

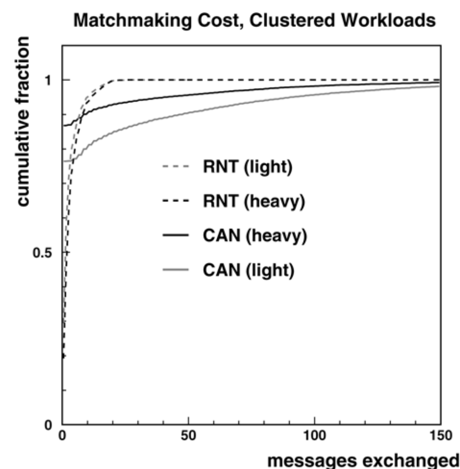
Our metrics are *matchmaking cost* (the number of messages required for finding candidate run nodes by the owner node of a job), *wait time* (the amount of time between when a job is injected and when it actually starts running), and *average queue length*, which is the length of the non-preemptive job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

We test the CAN approach (CAN, Section 3.2), RNT approach (RNT, Section 3.3), and the idealized centralized approach (Centralized, Section 3.4) that uses up-to-date global information to choose the node with the shortest queue length from all nodes in the system. We do not include "matchmaking cost" numbers for the centralized approach because it requires no messages.

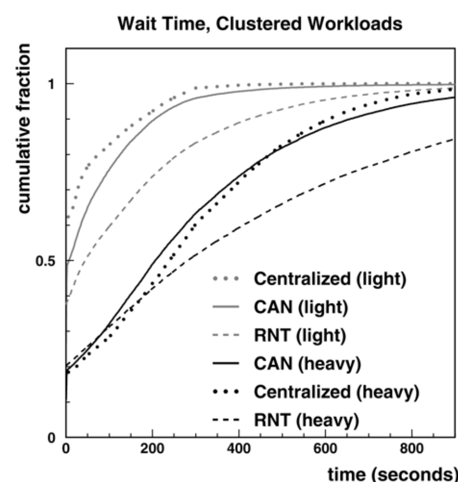
4.2. Experimental results

Fig. 2 shows matchmaking cost (messages), wait time, and queue length for the clustered workloads, while Fig. 3 shows the corresponding data for mixed workloads. For the clustered workloads, the RNT has lower matchmaking costs, but CAN has lower wait times and smaller queue lengths. The difference in queue lengths explains the difference in wait times, and comes from the virtual dimension allowing the nodes in a cluster to be spread through the CAN space. More specifically, in the clustered workloads, many nodes have identical resource capabilities so that the overall CAN space is split along the virtual dimension. This results in *coarse-grained* ranges in the real dimensions, where each node maintains large zones relative to its own resource capabilities. Therefore, matchmaking in CAN becomes expensive for jobs that have a small number of very high resource requirements. However, for jobs that have more constraints, overall matchmaking performance is better since jobs with many constraints are more likely mapped to the right region in the space where many candidate run nodes are available. However, contrary to the coarse-grained ranges in the real dimensions, the ranges for virtual dimensions become *fine-grained*, which spreads similar jobs uniformly across multiple nodes in the system to achieve superior load balancing compared to RNT and close to Centralized (as seen in Figs. 2(b) and (c)).

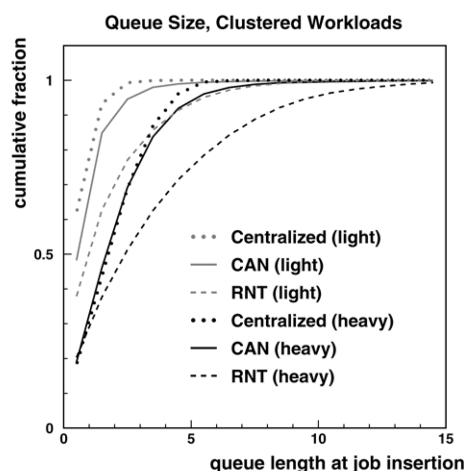
The mixed workloads provide a slightly different story. The matchmaking cost and the wait time for the "heavy" constraint workload still favor CAN, but CAN's performance on the "light" constraint mixed workload is much worse than that of RNT. Fig. 3(c) shows that queue lengths are much larger and more varied in CAN than in RNT, implying load imbalance. To understand why the resulting load imbalance



(a) Matchmaking cost.



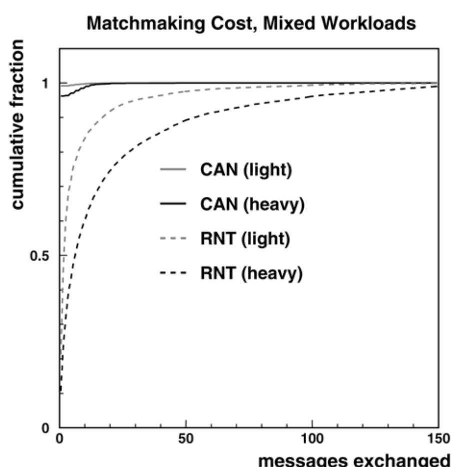
(b) Wait time.



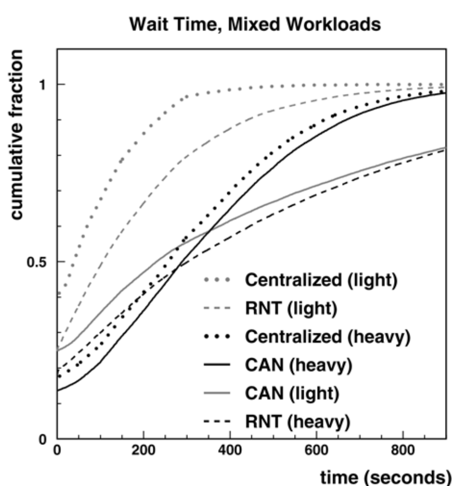
(c) Queue length.

Fig. 2. Performance results for clustered workloads.

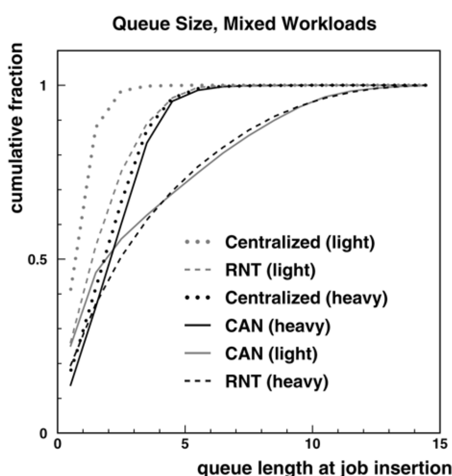
Author's personal copy



(a) Matchmaking cost.



(b) Wait time.



(c) Queue length.

Fig. 3. Performance results for mixed workloads.

is worse than in the clustered case, consider a hypothetical CAN with only a single real dimension, CPU speed. If most jobs do not specify CPU requirements (light constraint), their CPU speed coordinates will have the minimum value in that dimension. The jobs can still be mostly distributed (via the virtual dimension) along a line at a single CPU coordinate. However if most nodes have distinct CPU speeds (mixed node profiles), the slowest node ends up covering the bulk of the virtual dimension at low CPU speed, and will become the owner of a disproportionate number of the jobs, resulting in a *hot spot* and load imbalance.

Fig. 4 shows average wait times for three *light mixed dynamic* workloads. In these workloads, after 1000 nodes initially join the system, new nodes join and some existing nodes depart the system, which overall results in between 10% and 30% of the nodes eventually leaving during the course of the simulation (the Dynamic III has the highest node departure rate). Node departures are evenly split between graceful departures, where a node informs its neighbors before leaving, and failures, where the neighbors learn of the departure from the lack of heartbeat messages. For all the three dynamic workloads the number of jobs is about 10 000, which is similar to the static workloads, but different sets of nodes are available in the system at different times, so that we cannot directly compare across workloads.

The CAN and RNT approaches perform poorly relative to Centralized because of the need to recover and reconfigure the network. Although we cannot directly compare results across the three dynamic workloads (higher departure rates make the system less stable), the wait times are worse for CAN than for RNT or Centralized as the overall system becomes more unstable. Therefore, CAN's performance appears to be more affected than RNT's by increasing the departure rate. Since all of the dynamic workloads are based on mixed sets of nodes and jobs, a load imbalance problem similar to the one seen for the CAN earlier, due to a hot spot in the CAN space, can occur as jobs are entering the system and being assigned to run nodes. However, if one of the nodes in the hot spot leaves the system or fails, that can be disastrous for wait time performance, since all of the jobs that were running or waiting in the departed node must be reassigned to live nodes in the system. Since each node in the hot spot already has a disproportionate number of assigned jobs, this causes even more severe load imbalance for CAN-based matchmaking. However, in the RNT approach, since all of the jobs are assigned to owner nodes by a uniformly random function, it can achieve more even job allocations compared to CAN and is affected less by the dynamism of the system.

4.3. Discussion

The RNT and CAN algorithms have different underlying rationales. The idea motivating the RNT approach is to balance load by randomizing job assignment, mitigating the cost of matching demanding jobs by passing static capacity information across the tree (*matchmaking after load balancing*). Job assignment essentially consists of a randomized mapping, followed by a tree traversal to find a lightly loaded node capable of running a given job (i.e., meet the minimum resource requirements of the job). The idea behind the CAN approach is to first find a node whose capabilities approximately match the job's constraints, followed by a local

Author's personal copy

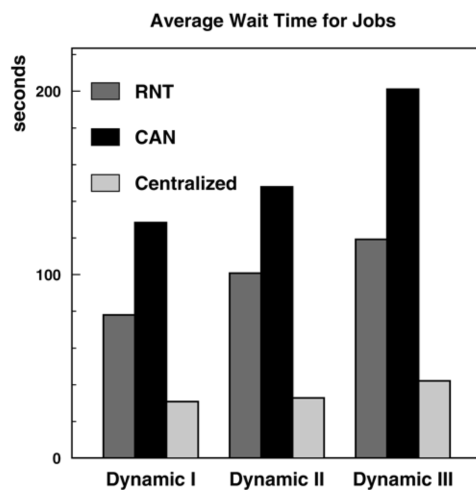


Fig. 4. Dynamic workloads.

search among similar nodes to find one that is lightly loaded (*load balancing after matchmaking*).

Both the RNT and CAN algorithms can cause poor load balance in at least two ways. First, the search path (a tree traversal for RNT and a local search for CAN) may not be long enough to find existing lightly loaded nodes. However, that may be a less serious problem for the CAN approach because each CAN node stores a limited load information for neighbor nodes. A second potential cause of load imbalance is poor matches between jobs and nodes (i.e, poor *Precision*). RNT can be thought of as a *first-fit* algorithm; it selects as the run node the most lightly loaded of a set of randomly chosen nodes, such that each node meets the minimum job constraints. However, the chosen run node might be greatly over-provisioned for the job, and this over-provisioning might not be useful. For example, over-provisioning in terms of CPU rate may be useful because it can speed up the execution of a given job, but an extra GB of memory might not improve execution time, and therefore not be useful. Meanwhile, other jobs needing the extra memory might be needlessly queued. In contrast, CAN is more of a *best-fit* algorithm (more precise) because the search starts at the node most closely matching the job's constraints.

Dynamism of the system also can affect the performance of CAN and RNT matchmaking mechanisms. Because existing nodes depart the system, the information carried by the CAN- and RNT-based mechanisms can be stale compared to the information maintained for static workloads, and there can also be some overhead for P2P network recovery. Additionally, reliable job assignments become more critical in dynamic environments, as seen from the results for the CAN approach, where the hot spots in the light mixed workloads become a problem for load balancing.

5. Related work

Recently there have been several research efforts to combine P2P and Grid computing techniques to improve the robustness, reliability and scalability of client–server-based desktop grid systems [17].

Several groups [18,19] have proposed P2P architectures to locate and allocate resources in a Grid environment employing a *Time-To-Live* (TTL) mechanism. TTL-based mechanisms are relatively simple but effective to find a resource (that meets the job constraints) in a widely distributed environment without incurring too much overhead in the search. However, such mechanisms may fail to find an appropriate resource on which to run a given job, even though such a resource exists somewhere in the network, because of the TTL mechanism (lack of *Completeness*).

Studies on encoding static or dynamic information about computational resources using a DHT hash function for resource discovery have also been conducted [20–24,14]. Research such as [20–22,24] employs one DHT for each resource attribute and performs matchmaking for the multi-attribute queries based on either controlled flooding [20] or sequential search [21,22], both of which have shortcomings with respect to search performance when there are a large number of resource attributes (lack of *Low overhead*). Registering all resource attributes in a single DHT, which enables efficient matchmaking [14,23], can negatively impact load balancing. A small fraction of the nodes might contain a majority of the resource information whenever there are many nodes with very similar (or identical) resource capabilities in the system (lack of *Load balance*). Also, simple encoding of resource information cannot effectively avoid selecting resources that are over-provisioned with respect to the jobs (lack of *Precision*).

The CCOF (Cluster Computing on the Fly) project [15,25] conducted a comprehensive study of generic searching methods in a highly dynamic P2P environment to locate idle computer cycles throughout the Internet. More recent work from the CCOF group on a peer-based desktop grid system called WaveGrid, constructed a *timezone-aware* overlay network based on CAN to use idle night-time cycles geographically distributed across the globe [16]. However, the host availability model in that work is not based on the resource requirements of the jobs nor the varying capabilities of nodes in the system (lack of *Capability*).

Cappello et al. proposed a computational Peer-to-Peer system called XtremWeb [26], whose aim is to investigate issues in turning a large scale distributed system into a parallel computer. The system provides user, administration and programming interfaces that could be used to harness simultaneously uncoordinated set of resources. However, the *coordinator* component in the XtremWeb architecture, which performs mediation between *clients* and *workers*, is implemented in a centralized way so is not scalable and robust as in a P2P-based desktop grid system.

6. Conclusions

In this paper we have described two different approaches that use P2P protocols to provide job scheduling and resource matching facilities for desktop grids. The CAN algorithm produces significantly lower wait times than the RNT approach over a broad spectrum of workloads. The result is that the

Author's personal copy

CAN approach is both more flexible and more efficient, for the general case where the workload has a great deal of diversity. However, CAN's poor performance with the "light mixed" workload is indicative of a broader problem in the robustness of the CAN load balancing. While the virtual dimension feature helps to smooth clumpy job and node distributions, thereby enabling better matchmaking, it is not always sufficient.

We are addressing this problem in our ongoing work by allowing the CAN matchmaking mechanism to *push* jobs into underloaded regions of the CAN space [27]. The decision about whether to push a job uses dynamically aggregated load information; a fixed amount of current system load information is propagated along each dimension in the CAN space. If the overall system is lightly loaded, jobs can also be pushed into the upper regions of the CAN space, so as to use more capable nodes in the system. Spreading out jobs via the push mechanism should allow the CAN algorithm to achieve better load balance for job assignments in dynamic environments.

Our work up to now has mainly considered *continuous* constraints for a job, such as minimum required CPU speed and memory size. However, we must also deal with *discrete* constraints for a job, such as operating system type and version. These kinds of discrete constraints can make the matchmaking process more difficult, since we have to find both exact matches for discrete constraints and approximate matches for continuous constraints in a single protocol. Also, by introducing more dimensions in the CAN space, the overall performance of matchmaking and load balancing can be affected. In particular, higher dimensions tend to result in shorter paths (both for CAN routing and for matchmaking) but a greater amount of state to store at a node. Addressing discrete constraints and their impact on performance is a subject of future work.

We are in the process of building a prototype system using CAN-based matchmaking, and will characterize its behavior on real workloads, via consultation with our application-area collaborators in astronomy and physics. In the future, we will measure and report on the behavior of our system for heterogeneous environments running real applications.

References

[1] A. Chien, B. Calder, S. Elbert, K. Bhatia, Entropia: Architecture and performance of an enterprise desktop grid system, Journal of Parallel and Distributed Computing 63 (5) (2003) 597–610.

[2] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer, SETI@home: An experiment in public-resource computing, Communications of the ACM 45 (11) (2002) 56–61.

[3] D.P. Anderson, C. Christensen, B. Allen, Designing a runtime system for volunteer computing, in: Proceedings of the 2006 IEEE/ACM SC06 Conference, 2006.

[4] A. Iamnitchi, D. Talia, P2P computing and interaction with grids, Future Generation Computer Systems 21 (3) (2005) 331–332.

[5] J. Ledlie, J. Schneidman, M. Seltzer, J. Huth, Scooped, again, in: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems, IPTPS '03, 2003.

[6] J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, A. Sussman, Resource discovery techniques in distributed desktop grid environments, in: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, GRID 2006, 2006.

[7] M.J. Litzkow, M. Livny, M.W. Mutka, Condor — A hunter of idle workstations, in: Proceedings of the 8th International Conference on Distributed Computing Systems, 1988.

[8] D. Epema, M. Livny, R. van Dantzig, X. Evers, J. Pruyne, A worldwide flock of condors: Load sharing among workstation clusters, Future Generation Computer Systems 12 (1) (1996) 53–65.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content addressable network, in: Proceedings of the ACM SIGCOMM, 2001.

[10] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of the ACM SIGCOMM, 2001.

[11] J.-S. Kim, B. Nam, M. Marsh, P. Keleher, B. Bhattacharjee, D. Richardson, D. Wellnitz, A. Sussman, Creating a Robust desktop grid using peer-to-peer services, in: Proceedings of the 2007 NSF Next Generation Software Workshop, NSFNGS 2007, 2007.

[12] C. Tang, Z. Xu, S. Dwarkadas, Peer-to-peer information retrieval using self-organizing semantic overlay networks, in: Proceedings of the ACM SIGCOMM, 2003.

[13] J.-S. Kim, B. Bhattacharjee, P.J. Keleher, A. Sussman, Matching jobs to resources in distributed desktop grid environments, Tech. Rep. CS-TR-4791 and UMIACS-TR-2006-15, University of Maryland, Department of Computer Science and UMIACS, 2006.

[14] D. Oppenheimer, J. Albrecht, D. Patterson, 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, 2005.

[15] D. Zhou, V. Lo, Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system, in: Proceedings of the 4th International Workshop on Global and Peer-to-Peer Computing, 2004.

[16] D. Zhou, V. Lo, WaveGrid: A scalable fast-turnaround heterogeneous peer-based desktop grid system, in: Proceedings of the 20th International Parallel & Distributed Processing Symposium, 2006.

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

[18] A. Iamnitchi, I. Foster, A peer-to-peer approach to resource location in grid environments, in: J. Nabrzyski, J.M. Schopf, J. Weglarz (Eds.), Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Publishers, 2004, pp. 413–429.

[19] A.R. Butt, X. Fang, Y.C. Hu, S. Midkiff, Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing, in: Proceedings of the 3rd Virtual Machines Research and Technology Symposium, VM'04, 2004.

[20] A. Andrzejak, Z. Xu, Scalable, efficient range queries for grid information services, in: Proceedings of the 2nd International Conference on Peer-to-Peer Computing, 2002.

[21] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: Supporting scalable multi-attribute range queries, in: Proceedings of the ACM SIGCOMM, 2004.

[22] M. Cai, M. Frank, J. Chen, P. Szekely, MAAN: A multi-attribute addressable network for grid information services, in: Proceedings of the 4th IEEE/ACM International Workshop on Grid Computing, GRID 2003, 2003.

[23] A.S. Cheema, M. Muhammad, 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, 2005.

[24] R. Gupta, V. Sekhri, A.K. Somani, CompuP2P: An architecture for internet computing using peer-to-peer networks, IEEE Transactions on Parallel and Distributed Systems 17 (11) (2006) 1306–1320.

Author's personal copy

[25] V. Lo, D. Zhou, D. Zappala, Y. Lin, S. Zhao, Cluster computing on the fly: P2P scheduling of idle cycles in the internet, in: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems, IPTPS '04, 2004.

[26] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Nri, O. Lodygensky, Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid, Future Generation Computer Systems 21 (3) (2005) 417–437.

[27] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, 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, 2007.

Jik-Soo Kim is a research assistant in the Department of Computer Science at the University of Maryland, College Park. Kim received a B.S. (1997) and an M.S. (1999) from the Seoul National University, Republic of Korea. Kim received an M.S. in Computer Science (2005) from the University of Maryland at College Park. Kim's research focuses on Peer-to-Peer (P2P) and Grid computing systems.

Beomseok Nam is a post-doc research associate at the University of Maryland, College Park. Nam received his Ph.D. in Computer Science in 2007 at the same university, and obtained a B.S. (1997) and an M.S. (1999) from Seoul National University, Republic of Korea. His research interests are in the area of distributed multidimensional indexing, P2P and Grid computing systems.

Peter J. Keleher received a Ph.D. in computer science from Rice University in 1995. He is currently an Associate Professor in the Computer Science Department at the University of Maryland, College Park. Professor Keleher's primary interests are in the design and analysis of distributed computing infrastructure, distributed security infrastructure, and communication performance.

Michael A. Marsh is an assistant research scientist at the University of Maryland Institute for Advanced Computer Studies. Marsh has a B.A. from Cornell University and an M.S. and Ph.D. (2001) from the University of Illinois at Urbana-Champaign. Marsh's research focuses on information assurance in distributed systems, with an emphasis on establishing and maintaining trustworthiness and reliability.

Bobby Bhattacharjee is an associate professor in the Computer Science department at the University of Maryland, College Park. His research interests are in the design and implementation of scalable systems, protocol security, and peer-to-peer systems. He is a member of the ACM, and a fellow of the Sloan Foundation.

Alan Sussman is an Associate Professor in the Computer Science Department at the University of Maryland, College Park. His research interests include Peer-to-Peer (P2P) systems, high performance database and I/O systems, coupled multiphysics simulations, Grid computing, and runtime environments for distributed and parallel systems. He received his Ph.D. in computer science from Carnegie Mellon University in 1991 and his B.S.E. in Electrical Engineering and Computer Science from Princeton University in 1982.