Matchmaking and Implementation Issues for a P2P Desktop Grid

Michael Marsh[1], Jik-Soo Kim[1], Beomseok Nam[1], Jaehwan Lee[1], San Ratanasanya[1],
Bobby Bhattacharjee[1], Peter Keleher[1], Derek Richardson[2], Dennis Wellnitz[2] and Alan Sussman[1]
[1]UMIACS and Department of Computer Science
[2]Department of Astronomy
University of Maryland, College Park, MD 20742
[1]{mmarsh,jiksoo,bsnam,jhlee,san,bobby,keleher,als}@cs.umd.edu
[2]{dcr,wellnitz}@astro.umd.edu

Abstract

We present some recent and ongoing work in our decentralized desktop computing grid project. Specifically, we discuss matching jobs with compute nodes in a peer-to-peer grid of heterogeneous platforms, and the implementation of our algorithms in a concrete system.

1 Introduction

Centralized desktop grid systems [1, 2] face inherent bottlenecks in the management nodes. Consequently, researchers have been investigating decentralized grid architectures, built largely on peer-to-peer designs [3, 8].

The loss of centralization makes the task of matching jobs to computational nodes capable of meeting the jobs' requirements considerably more difficult. In addition, the implementation issues faced by grids generally are compounded by the addition of a peer-to-peer layer. In this paper we outline the issues and challenges that have arisen in own peer-to-peer desktop computational grid system, targeted at allowing a set of collaborating users to share computational resources.

Architecture Overview In previous work [4, 5, 6], we found that a Content-Addressable Network (CAN) [7] provides a good framework for a decentralized grid. A CAN is a type of distributed hash table (DHT) that maps nodes and jobs into a multidimensional space. In our case, nodes are mapped by their resource capabilities (each resource type is a separate dimension), and jobs by their resource requirements. The CAN algorithms partition the space into non-overlapping (hyper-rectangular) regions, with each node responsible for one region. The semantics of forwarding in a CAN places a job at a node that is minimally capable of running that job. (More likely, due to the geometry of the multidimensional space, the node is *almost* capable of running the job.) The further task of choosing a node to run the job proceeds from that point in the space.

2 Matchmaking

2.1 Goals

A general-purpose desktop grid system must accommodate various combinations of node capabilities and job requirements. Nodes may be added one at a time over time, so that their resource capabilities are heterogeneously distributed, or they may be added as homogeneous clusters. Likewise, jobs may be relatively unique in their requirements, or part of a series of requests with similar or identical requirements (e.g., a parameter sweep application).

The implication is that a matchmaking algorithm must incorporate both node and job information (resource capabilities and requirements, respectively) into the process that eventually maps a job onto a specific node. To be able to handle *general* configurations of nodes and jobs, the goals of any matchmaking algorithm should meet the following criteria:

1. *Expressiveness* - The matchmaking framework should allow users to specify minimum (or exact) requirements for any type of resource
2. *Load balance* - Load (jobs) must be distributed across the nodes capable of performing the work.
3. *Parsimony* - 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.

2.2 Matchmaking Overview

2.2.1 The Easy Case

Matchmaking begins at the node closest in capabilities to the requirements of the job. This node, the *owner*, looks at the information it has for its nearest neighbors and next-nearest neighbors in the partitioned multidimensional space that satisfy the job's requirements. If any of these has an empty queue, it is selected as the *run node* for the job.

2.2.2 Extending the Horizon

In an active grid system, there is not likely to be a neighbor of the owner without any jobs in its queue. Consequently, we need a way to extend our view of available nodes beyond those for which we have direct information.

The Setup The normal maintenance of a CAN involves regular exchange of messages between nodes. We augment these messages with a small amount of information, related to overall system load. In particular, we record an approximation of the number and load of nodes extending towards the resource capability maxima in each dimension of the CAN. As messages are exchanged, this information flows from the more capable nodes down to the less capable nodes in each dimension, aggregating the information from intervening nodes along the way. Thus we build an approximate view of the upper regions of the CAN space.

Finding a Run Node Consider a node n that is processing a matchmaking request for the job. Initially, this is the job's owner node. n looks at the information it has aggregated from its neighbors. Using some type of scoring function, it picks one of these neighbors that seems most likely to reach a free node quickly. The job is then forwarded to this neighbor, which repeats the process. We refer to this as "pushing" the job. Because information is aggregated toward the CAN origin, jobs are pushed to more capable nodes. If a job ever reaches a node with an empty queue (and that is capable of running the job), that node becomes the run node. We discuss more details of the difficulties in this part of the problem later. Figure 1 depicts the overall process of finding a run node after a client submits a job.

2.3 Challenges

Aggregation The first challenge with load aggregation is what, precisely, to aggregate. From simulations, we find that using the number of nodes and sum of the job queue lengths allows us to balance the load fairly effectively.

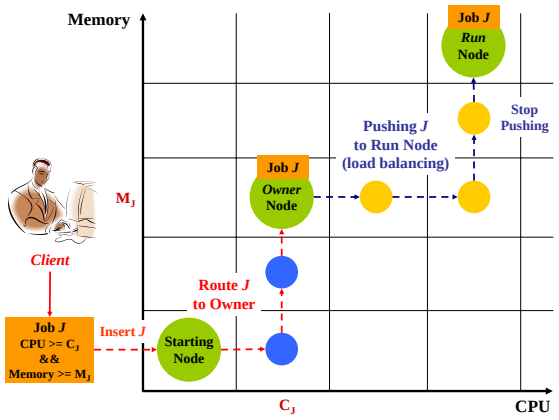

Figure 1. Matching a Job to an Available Capable Node

The second challenge is *how* to aggregate this information. The simplest and most obvious choice is to aggregate along the individual dimensions. That is, a node n has aggregated information from its neighbor with more disk space regarding all other nodes in a straight line from n outward in the disk space dimension. Unfortunately, this misses the possibility that a node slightly off this line has a very short job queue. We have been investigating ways of capturing this off-axis load information, generally involving attenuating information in proportion to its distance from the axis along which it is aggregated.

The scoring function, on which we base our pushing decision, is another issue. The straightforward solution uses the average queue length of the nodes in each CAN dimension. Again, this misses some relevant features for load distribution. That is because nodes in the dimension with the shortest average queue length are not always the best choice. Routing in a dimension with a slightly longer average queue size might enable access to a larger number of potential run nodes than the dimension with the smallest average queue length. Seeing the larger number of nodes makes it more likely that when a pushed job reaches one of the nodes believed to be lightly loaded, that node will still be lightly loaded. Therefore, our scoring function considers both average job queue length and the number of available nodes propagated during load aggregation.

One final challenge regarding aggregation is that the information rapidly becomes stale. There is no way to avoid this, except by more frequent message exchanges. Recent work on our system has employed partial exchanges, where only some neighbors receive updates. While this improves many features of the system, especially its ability to scale, it also exacerbates the aggregated load staleness problem.

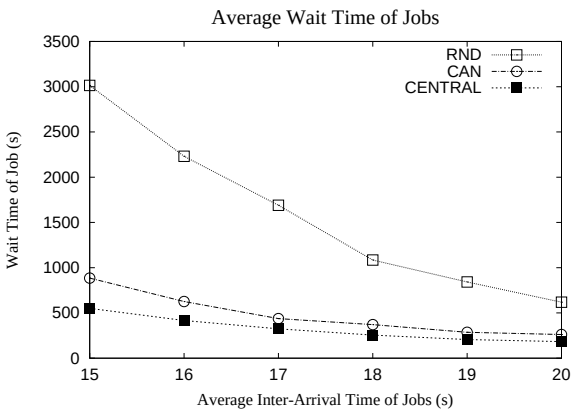

Figure 2. Load Balancing Performance of the CAN algorithms

Pushing It is not enough to aggregate load information – we must also be able to use it to move jobs towards lightly loaded potential run nodes. Because pushing, as we envision it, involves sending a single copy of the job description out from the owner node, matchmaking is susceptible to message losses. These might be due to network failures, data corruption, or messages discarded by an errant node. Some sort of mechanism to detect and recover from this situation is necessary. As a last resort, the client that submitted the request might time out waiting for a response indicating that the job has been enqueued, prompting it to resubmit the job. This, however, is an unsatisfying solution.

The larger challenge when pushing a job is deciding when to *stop* pushing it. The job may never reach a node with an empty queue, which means we need some mechanism for a node to decide, based solely on local information, that it will not push the job further, but rather nominate itself as the run node. We have investigated a probabilistic mechanism for this purpose, based on the estimated number of nodes reachable by pushing from the current node. When establishing this stopping probability, we need to balance the likelihood of finding a better choice by continued pushing against the needs of legitimately resource-demanding jobs which have a smaller pool of potential run nodes.

2.4 Performance Results from Simulations

Figure 2 shows experimental results obtained via event-driven simulations. All results were collected when the system was in the *steady state*, where the rate for jobs arriving and completing is approximately the same. As the average inter-arrival time between jobs increases, the overall system becomes less loaded. To see how well the workload could be balanced, we measure average job wait time in the job queue on a node and compare the CAN-based approach (**CAN** in the figure) with an idealized online *centralized* scheme (**CENTRAL**) that uses knowledge of the status of all nodes and jobs, so cannot be efficiently implemented in a decentralized system. We also show results for a relatively simple *randomized* approach (**RND**) that achieves load balancing based on random assignment of jobs to the nodes and meets the jobs' constraints through a local search (for more details on RND and CENTRAL, see [4]). As the results show, the CAN algorithms outperform RND, especially under high load, and balance load almost as well as CENTRAL. In results not shown due to lack of space, we also verified that the cost for performing matchmaking in the CAN is very low compared to the time to run a job.

3 Implementation

3.1 Goals

As a real-world computational grid, the system must be *long-running*. That is, we should not expect the entire grid to be brought down and restarted except under exceptional circumstances. One thing that works in our favor is that the system is completely decentralized, so individual failures have a minor impact on the system as whole, provided the mean time to failure for any participant is large.

Our system is also designed to have a small impact on the perceived performance of the hosts on which it runs. For desktop grid participants, local users' tasks must run with highest priority. Beyond that, grid jobs should consume as much processing time as they reasonably can. The operation and maintenance of the grid software should have *minimal impact* on the participants. Ideally, the CPU and network usage should be unnoticeable. Minimizing the system's impact on hosts cannot be the sole concern, however, since users will expect the system to be *responsive*. Researchers should not have to wait five minutes to find out whether jobs they have submitted have been enqueued successfully.

Finally, the resource providers and job submitters are generally assumed to be well-intentioned. However, the grid *will* be visible to the Internet, which includes many malicious individuals. Moreover, we cannot expect submitted jobs to be bug-free, whether in implementation or logic. Consequently, we need to ensure that the system is *"reasonably" secure*.

3.2 Overview

3.2.1 CAN

The underlying peer-to-peer architecture is a highly customized implementation of a Content-Addressable Network

(CAN). A standard CAN has a number of dimensions with no semantic meaning. Effectively, each is an independent hash of some unique characteristics of the node or data. The number of dimensions is chosen to balance per-node state and routing costs. However, our design and implementation depends on dimensions corresponding to the resource types provided by nodes or required by jobs, as well as a single *virtual* dimension that more closely corresponds to the original CAN design. Since the point of the virtual dimension is to distinguish between nodes that are otherwise identical and help spread jobs over these nodes, rather than using a hash function we randomize the values of nodes and jobs in the virtual dimension.

Our implementation is designed to run on non-dedicated hosts. That is, we do not expect either our system or the jobs it runs to be of great importance to the principal user of a participating machine. That means the grid application must be easy to run, unobtrusive, and robust.

3.2.2 Jobs

The purpose of the grid is to run jobs submitted by users. These jobs will come from users that may or may not be from the same department or even institution as the hosts on which they will run. Since job executables and data files might be large, we do not move these around in the system. Rather, we pass *descriptions* of jobs, which include instructions on how to retrieve the needed files and how to compose them in a job invocation.

Once placed at a run node, jobs are processed in FIFO order; there is no notion of job priorities. Each job runs in its own directory, with all of its input and output files. Both the standard output and standard error streams are captured in files, though the job is also free to create additional output files. When a job completes, the run node creates a `tar` file of the job's directory and sends it to the client via a standard POSIX TCP socket.

3.3 Challenges

3.3.1 CAN

A review of the literature shows that few implementations of CAN exist. This is because CAN is significantly more difficult to implement than other distributed hash table algorithms. We compound the difficulty by creating a version of CAN that is even *more* difficult to implement.

Our semantics mean that the standard CAN technique for inserting a new node, which is to split the current zone containing the node's multidimensional point evenly along its longest current dimension, is not possible. Consequently, we may have zones that cover a large range in one dimension, but very small ranges in the others. This is further exacerbated by the fact that we are unable to balance node placement through uniformly random placement in the space.

Zone management is also more complicated. Holes are not permitted in the CAN space, either for the original design or our implementation. The original design does not make any intimate connection between a node and the region of the space it covers, so it is acceptable for a single node to maintain two distinct rectangular zones. In our case, this does not work, since zone ownership implies a certain minimum capability in terms of real resources. The failure recovery mechanisms in the original CAN therefore become brittle in our system. We have, however, solved all of these problems in our design and implementation, and verified their correctness and adequate performance characteristics through simulations.

To add another complication, we intend the system to run on any POSIX host. Hence we chose to restrict our implementation to the C99 standard of the C programming language, with the addition of POSIX features. By restricting ourselves in this way, our source code should be readily portable to additional systems beyond those on which we are currently developing (Linux, Solaris and OSX). The GNU *autoconf* package gives us the ability to compensate for the remaining variations between platforms.

3.3.2 Keeping Sysadmins Happy

A crucial aspect of a grid computing system is that it has to run on hosts for which system administrators are responsible. These sysadmins tend to be conservative, so the implementation must be demonstrably not harmful.

Network Network usage affects all users, and a substantial slowdown will result in a flood of complaints. This will, understandably, make a sysadmin ill-disposed to continued participation in the grid. The implication of this is that basic grid maintenance messages, such as job heartbeats and periodic neighborhood exchanges, must be kept as small as possible, lest they give the appearance of malware. The frequency of message exchanges must also be limited, since this too can overburden a network. Message frequency requires trading off bandwidth consumption against staleness of data. If we expect jobs to run for times on the order of hours, then periodic exchanges on the order of minutes should suffice. We are currently working on techniques to limit message sizes even for very large and arbitrarily zoned grids.

Hosts A user contributing his or her desktop machine to the grid will not appreciate a dramatic slowdown in responsiveness. Additionally, many hosts have network-mounted filesystems. Over-use of CPU (easily mitigated with the

`nice` command) or storage space are bad enough, operationally, but the potential for malicious behavior (including possible privilege escalation) is a major concern.

We address these issues first by restricting who can submit jobs to the grid. By using standard X.509 certificates, with a grid "owner" as the certification authority (CA), we cryptographically ensure that jobs are legitimate. We also use certificates to prevent malicious individuals from contributing hostile nodes to the grid. Not all messages need to be signed; established connections can be trusted to pass only legitimate traffic, so long as we discount the possibility that hosts are compromised.

Even with trusted users, the grid is running untrusted code that might contain errors in logic or implementation. Our main concern here is to limit the damage that buggy code might cause. The simplest protection that we can offer is the Unix `chroot` command. We use this, when available, to run jobs in a restricted environment that shields the bulk of the host's filesystem from the job. This, however, requires superuser privileges to set up. While we provide a readily auditable program to establish the jail, not all sysadmins will be willing to allow it. A more complex option is to use virtualization software, such as VMWare (*http://www.vmware.com*) or Xen (*http://xen.org*). This requires greater effort to set up, but can in principle be done by an ordinary user, and is completely independent of our implementation. We would like to provide the added protection of blocking jobs from accessing the network, but it is exceedingly difficult to keep user code from opening sockets.

3.3.3 Keeping Users Happy

A computational grid that works flawlessly is not useful unless researchers are willing and able to use it. Consequently, we have put considerable effort into the two components with which users will interact directly. There is also extensive documentation, in the form of a user guide.

Researchers will primarily interact with the grid through the client interface. This is a graphical application, built in C using the Gtk toolkit (*http://www.gtk.org/*), that allows users to submit jobs and monitor their progress. A simple text-based client is also available, which is suitable for scripted interactions with the grid.

The other major user-interface component is a Java application to generate job description files. The files themselves are in a custom XML format, which is designed to be fairly human readable. The job file generator allows researchers to combinatorially construct potentially large numbers of jobs, such as by specifying multiple values for each of several parameters in a simulation, or providing a set of data files to be processed identically by the same user program.

4 Summary

Decentralizing the management of a desktop computing grid brings with it many challenges, but it also brings increased scalability and resilience to failures. As we have encountered algorithmic challenges, we have found that modifications to our initial algorithms have been necessary, to maintain scalability and reliability. Further, while starting from an inherently complex distributed hash table, the implementation has never suffered from apparently insurmountable obstacles, nor any sacrifice in software design. The implementation is nearly ready for field testing, with the matchmaking algorithms still in the process of being refined. However, the parts of the implementation that have been validated by simulation are already in, or currently being added to, the peer implementation.

References

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

[2] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63(5):597–610, May 2003.

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

[4] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)*, June 2007.

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

[6] J.-S. Kim, B. Nam, M. Marsh, P. Keleher, B. Bhattacharjee, D. Richardson, D. Wellnitz, and A. Sussman. Creating a Robust Desktop Grid using Peer-to-Peer Services. In *Proceedings of the 2007 NSF Next Generation Software Workshop (NSFNGS 2007)*, Mar. 2007.

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

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