ABSTRACT

Title of Dissertation: DECENTRALIZED AND SCALABLE RESOURCE MANAGEMENT FOR DESKTOP GRIDS

Jik-Soo Kim, Doctor of Philosophy, 2009

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

The recent growth of the Internet and the CPU power of personal computers and workstations enables *desktop grid* computing to achieve tremendous computing power with low cost, through opportunistic sharing of resources. However, traditional server-client Grid architectures have inherent problems in robustness, reliability and scalability. Researchers have therefore recently turned to Peer-to-Peer (P2P) algorithms in an attempt to address these issues.

I have designed and evaluated a set of protocols that implement a scalable P2P desktop grid computing system for executing Grid applications on widely

distributed sets of resources. Such infrastructure must be decentralized, robust, highly available and scalable, while effectively mapping application instances to available resources throughout the system (called *matchmaking*).

First of all, I address the problem of efficient matchmaking of jobs to available system resources by employing customized Content-Addressable Network (CAN) where each resource type corresponds to a distinct dimension. With this approach, incoming jobs are matched with system nodes through proximity in an N-dimensional resource space. Second, I provide comprehensive load balancing mechanisms that can greatly improve overall system throughput and response time without using any centralized control or information about the system. Finally, to remove any hot spots in the system where a small number of nodes are processing a lot of system maintenance work, I have designed a set of optimizations to minimize overall system overheads and distribute them fairly among available system nodes. My ultimate goal is to ensure that no node in the system becomes much more heavily loaded than others, either because of executing jobs or from system maintenance tasks. This is because every node in our system is a peer, so that no node is acting as a pure server or a pure client.

Throughout extensive experimental results, I show that the resulting P2P desktop grid computing system is scalable and effective so that it can efficiently match

any type of resource requirements for jobs simultaneously, while balancing load among multiple candidate nodes.

DECENTRALIZED AND SCALABLE RESOURCE MANAGEMENT FOR DESKTOP GRIDS

by

Jik-Soo Kim

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Advisory Committee:

Professor Alan Sussman, Chairman/Adviser
Professor Peter J. Keleher, Co-Adviser
Professor Bobby Bhattacharjee
Professor Atif M. Memon
Professor Derek C. Richardson

DEDICATION

To my wife – Boram Lee for her love and support.

ACKNOWLEDGEMENTS

My gratitude to those who have helped me to complete this dissertation cannot be adequately expressed here. Please accept my apologies if you find yourself unjustly missing or find your contribution inadequately credited. I really appreciate all those who supported me in one way or another.

I have been very fortunate to work with my thesis adviser, Dr. Alan Sussman, whose suggestions led me throughout this dissertation. He has encouraged me in all the time of research and has guided me on the right track with his penetrating insight to all the problems I had. I am indebted to my former colleague and friend, Beomseok Nam who helped me to build a cornerstone for my research.

Several faculty members provided me with their guidance about my proposal and feedback about this work, including Dr. Peter Keleher,

Dr. Bobby Bhattacharjee, Dr. Atif Memon, and Dr. Derek Richardson. Especially, I would like to give my special thanks to Dr. Peter Keleher who has co-advised me through all the way of my research.

I am very grateful to my close friends, Il-Chul Yoon, Youngmin Kim and Minkyoung Cho who came to the College Park with me at the same time. Because of them, I could have completed such a long journey of graduate school without any major problems. Also, I have been very happy to meet all of our KGCS members who gave me such a great time. Especially, I was fortunate to have our group members, Jaehwan Lee and Sukhyun Song and wish everything goes well with them all the time.

Finally, I cannot express my great gratitude in any words to my wife Boram Lee and my parents for their love and support.

January, 2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The recent growth of the Internet and the hardware capability of personal computers and workstations enables distributed computing to achieve tremendous computing power by harnessing a large number of machines. These systems are often called *desktop grid* computing systems and leverage unused capacity of high-performance desktop PCs [3, 4, 5, 19, 26, 34, 93]. 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. Existing architectures for desktop grid computing are typically based on a client-server model, where a trusted server supplies jobs to a set of client machines distributed across the Internet. Robustness and reliability are guaranteed by the server maintaining the status of all outstanding jobs running on poten-

tially 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 be reliable, otherwise the status of outstanding jobs could be lost. The server typically stores the state of jobs in a (relational) database, which provides some level of reliability. However, this centralized client-server architecture is vulnerable to a single point of failure, i.e., no new jobs can be assigned to a client whenever the server becomes unavailable either due to server failure or network partition. Also, the centralized server can easily become performance bottleneck, which results in inherent shortcomings with respect to robustness, reliability and scalability. Finally, since only the desktop grid server supplies jobs to the client machines across the Internet, existing desktop grid computing platforms do not allow arbitrary users to submit their own jobs to the pool of available computational resources.

Our goal is to design and build a scalable infrastructure for executing Grid applications on a widely distributed set of resources. Such infrastructure must be *decentralized, robust, highly available*, and *scalable*, while efficiently mapping application instances to available resources throughout the system (called *matchmaking*). Fortunately, these are precisely the characteristics promised by new techniques and approaches in Peer-to-Peer (P2P) systems. Using P2P ser-

vices 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, we can 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, essentially allowing a group of users to form an ad-hoc set of shared resources. The overall system, from the point of view of a user, can be thought of as a combination of a centralized, Condor-like grid system for submitting and running arbitrary jobs [27, 33, 61, 75, 89], and a system such as BOINC [3, 4] or SETI@Home [5] 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 [13, 25, 35, 36, 46, 47, 60].

However, how to apply P2P techniques to Grid computing area is not immediately obvious. Although both Grid and P2P systems have the same goals for global resource sharing, they have many essential differences imposed mostly by the behaviors and objectives of the involved users. Compared to Grid computing systems which are mainly used for complex scientific applications that are usually compute-intensive and require a large amount of resources, the most popular ser-

vice provided by P2P systems such as Gnutella [42] or Kazaa [50] is file sharing. Therefore, most of the research performed in the P2P community targets efficient locating and sharing data across a very large number of potentially unreliable machines, rather than running and monitoring Grid-like applications. To summarize, in order to appropriately employ P2P techniques in the Grid computing environment, we have to address several issues as follows:

1. *Job submission* - How can we submit a job into the decentralized and distributed environment?

2. *Matchmaking* - How can we find a resource that *meets* the resource requirements of a job without any centralized control and information about the system for better scalability?

3. *Load balance* - How can we distribute the load (jobs) across the nodes in the system?

4. *Resilience to failures* - The overall system must be resilient to failures of individual resources.

To address these issues, first I rely on the basic framework designed by our project team for submitting jobs, managing and monitoring jobs while they are

running, including methods for failure detection and recovery in a decentralized and distributed environment [56].

Second, I have developed decentralized and distributed resource management techniques in the P2P desktop grid computing system [52, 53, 54, 55, 57, 65]. The resource management algorithms include both efficiently matching jobs having different resource requirements with available heterogeneous computational resources and providing good load balancing to obtain high system throughput and low job turnaround times. However, as the overall system scales to large configurations and heavy workloads, it becomes a challenging problem to perform efficient matchmaking and load balancing all without any centralized control or information about the system. I address the problem of efficient matchmaking of jobs to available system resources by employing a customized Content-Addressable Network (CAN) [76], where each resource type corresponds to a distinct dimension. With this approach, incoming jobs are matched with system nodes through proximity in an N-dimensional resource space. Additionally, I provide effective load balancing mechanisms that can greatly improve overall system throughput and response time without using any centralized control or information about the system.

However, an effective P2P desktop grid system must ensure that no node in the

system becomes much more heavily loaded than others not only because of job executions but also from system maintenance. This is because every node in our system is a peer so that no node is acting as a pure server or a pure client. Therefore, to remove any hot spots in the system where a small number of nodes are performing a lot of system maintenance work, I have designed a set of optimizations to reduce overall system load and distribute it more fairly among available system nodes.

Throughput extensive experimental results, I show that the resulting P2P desktop grid computing system is truly scalable and effective so that it can efficiently match any type of resource requirements for jobs simultaneously, while balancing load among multiple candidate nodes.

1.1 Motivating Applications

Our target applications are usually compute-intensive but have relatively low I/O requirements. Examples of these applications include bioinformatics applications such as understanding protein folding, misfolding, and related diseases [34], Monte Carlo and other physical simulations in various scientific disciplines, and more esoteric applications such as searching for Mersenne primes [1] and search-

ing for extraterrestrial life [5]. However, unlike existing projects, the proposed system can allow arbitrary 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. Then the system should utilize all available computational resources to execute all submitted jobs in a fair manner. This includes allocating resources to requests both from users submitting a large number of jobs at once (as in a parameter sweep for a physical simulation application) and from users with smaller resource requirements. While some research work regards this resource allocation problem as a scheduling problem [22, 58], it is more appropriate to characterize it as one of fair resource allocation among peers because there is no centralized control or information about the system to assign resources to jobs [56].

With our astronomy collaborators at the University of Maryland, we have identified several problem areas with these characteristics in astronomy and physics, mainly related to physical simulations and data analysis such as follows:

Habitable Planets The scientific goal of this project is to determine where potentially habitable terrestrial planets might safely exist in extrasolar planetary systems. The orbital dynamics in systems containing three or more bodies (e.g., a star, a planet, and a test particle; or such a system with more planets; or a bi-

nary star system with more than one planet) are very complex and analytically intractable. To locate stable orbits for the test particle in such systems generally requires either sophisticated analytical estimates (which usually break down when the eccentricities of any of the massive bodies are large, as is the case in a number of extrasolar planet systems) or N-body integration techniques. For a given planetary system, we should compute at least 100,000 possible orbits for at least 10 million years to achieve a good impression of stable regions. A 10 million year integration of 250,000 test particles placed randomly on circular orbits in the asteroid belt region of our solar system was performed on the client-server grid system, and took several CPU months to complete. The proposed system can be used to explore the huge parameter space associated with this problem, and to obtain statistically meaningful results.

Formation of Asteroid Binaries Understanding the formation of asteroid binaries can show not only how asteroids evolve (which is essentially related to the formation of planets and therefore the origin of life), but also the internal structure of asteroids. Discovering that internal structure is an important requirement for developing hazard mitigation strategies if one of these bodies is headed toward the Earth and might result in a collision. If the current tidal disruption model of

asteroid binary formation is correct, most near-Earth asteroids are piles of rubble that cannot be efficiently destroyed with explosives, because the blast energy gets absorbed. However, there is an enormous parameter space of plausible encounter scenarios. The key variables include close-approach distance, encounter speed, progenitor spin state (both magnitude and orientation), shape elongation, and bulk density. This problem is ideally suited to the desktop grid model, since a single simulation involves a relatively small number of particles and no communication is needed between simulations. A study performed running 110,000 simulations where each takes over one hour on the fastest machine available so that indeed massively parallel executions of these simulations are imperative.

The Deep Impact Mission The NASA Deep Impact (*http://deepimpact.jpl.nasa.gov/home/*) mission team has identified several problems whose solutions require significantly more computing capabilities than are currently available to the team. One computational bottleneck is performing a large set of expensive deconvolution operations to correct data being returned from a faulty instrument on the spacecraft. The team has both measured and modeled a variety of deconvolution algorithms, but they are currently limited to relatively simple ones by the computational power available for the processing. Utilizing a larger number of machines in parallel

to address the optimal deconvolution for a particular application will provide considerably better scientific results faster. The deconvolutions are compute-intensive and involve many Fourier transforms on a single input image. Another computational challenge is theoretical modeling of spectra and photometry of the Deep Impact data. Currently the exploration of these models is limited by the amount of computational capability available. Like many small bodies, the nucleus of comet Tempel 1 is quite irregular, resulting in a need for good modeling of shape and illumination to account for the effects of these influences on the observed light intensities in various spectral filters and as a function of time. Similarly the variety of chemical species and their temperature states discernible with the medium resolution infrared spectrometer instrument is limited by the spectral modeling done so far. Most of this modeling is compute-intensive rather than I/O intensive, so that it can get benefits if a larger number of machines can be applied to the analysis.

All of these problem areas are examples of our target applications that are usually compute-intensive but require relatively low I/O operations. In this dissertation I create various synthetic workloads based on the characteristics of these applications rather than actually running them. Most of my work is implemented

and evaluated through event-driven simulations.

There are several reasons for performing simulations. First, we can test the behaviors of our proposed system under various scenarios of node capabilities and resource requirements of the jobs. Events include node joins, departures (graceful or failures), and job submissions so that we can inject the new nodes and jobs or generate node failures or departures at any time. The main reason for using synthetic workloads is that we could not find enough useful information throughout existing systems such as Condor [33, 61, 75, 89]. However, we expect as we deploy our real prototype system, we can collect more useful information about the job workloads. Second, we can easily verify the correctness of our algorithms for matchmaking and load balancing in a large set of node and job populations which would be very difficult in the real implementation to collect all the information for the analysis. Finally, we can effectively measure the performance and overhead of our proposed P2P desktop grid computing system. This is because it would be difficult to prove the functionality of our system theoretically in this dynamic decentralized environment.

We currently have a prototype peer implementation, and are in the process of characterizing its behavior on real workloads with large numbers of peers. We are working with our collaborators in physics and astronomy to deploy the desktop

grid system onto their machines, to enable them to share compute resources with colleagues across the globe. In the near future, we will measure and report on the behavior of the system in heterogeneous environments running real applications.

1.2 Thesis and Contributions

In this dissertation, I support the following thesis: *decentralized resource management can be employed to create scalable desktop grid computing systems*. Our system has two major advantages over the existing architectures for executing Grid applications on a widely distributed set of resources: *scalability* and *usability*. We make our system scalable by removing a single point of failure and contention so that it can scale gracefully as more nodes and jobs are injected into the system. Also, we provide an improved usability by allowing arbitrary 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. Therefore, as we mentioned earlier, the overall system, from the point of view of a user, can be thought of as a combination of a centralized, Condor-like grid system [33, 61, 75, 89] for submitting and running arbitrary jobs, and a system such as BOINC [3, 4] or SETI@Home [5] for

farming out jobs from a server to be run on a (potentially very large) collection of machines in a completely distributed environment.

By employing a centralized matchmaker [75], Condor system allows arbitrary users to submit and run their own jobs by specifying their resource requirements on the pool of available machines. However, there are two constraints that can limit Condor's potential of sharing available resources (i.e., scaling to a large number of resources). First, the centralized matchmaker is a single point of failure, and in case such a failure occurs, the whole pool becomes unavailable. Also, the centralized matchmaker is a single point of contention since as more nodes and jobs are inserted into the system, the amount of work performed by the matchmaker increases linearly. Second, the size of individual pools is limited by the resources available to an organization. This is because the Condor pool is intended to be deployed in a single administrative domain. Condor addresses the issue of sharing resources among multiple pools by a mechanism referred to as flocking [33]. However, this mechanism is static and requires manual configuration so that each user should be able to register in multiple different administrative domains and the matchmaking process can be shipped to another pool throughout the gateway machine. Again, if this gateway machine becomes unavailable, the flocking mechanism may not work properly. Our system can federate these multiple pools

of available machines by employing P2P techniques.

On the other hand, BOINC system employs a centralized server to deploy a larger number of independent jobs across available client machines on the Internet. This server again can become a single point of failure so that no new jobs can be assigned to a client whenever the server becomes unavailable either due to server failure or network partition. Clients connect the server and download the jobs to be computed voluntarily so that this type of computing systems is often called Volunteer Computing [4]. Therefore, the server in the BOINC system simply deploys jobs (which are dedicated to a specific project) across client machines and does not provide the matchmaking functionality. This means that systems such as BOINC do not allow arbitrary users to submit and run their own jobs on the pool of available machines which lacks of usability. However, our system provides more improved usability compared to the BOINC system by allowing arbitrary users to submit and run their own jobs.

To summarize, I define the "scalable" desktop grid computing system as the system having following properties:

1. The overall system is resilient to the failures unless there are multiple simultaneous failures that can break the structure of the system (no single point of failure)

2. The overall system scales gracefully as more nodes and jobs are inserted (no single point of contention). Our system's scalability is heavily based on the functionality provided by the distributed hash table, especially the Content-Addressable Network (CAN) [76]. The average routing path (which is closely related to the matchmaking cost) in the CAN is proportional to the number of dimensions (denoted as d) and $\sqrt[d]{N}$ where N is the number of nodes in the system [76].

To support this thesis, I develop, apply, and evaluate a set of techniques for building an effective and scalable P2P desktop grid computing system. More specifically, this dissertation makes the following contributions not discussed in previous related research:

1. **An efficient decentralized matchmaking framework**

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

over a large set of parameter combinations). A good matchmaking algorithm must be expressive enough to fully describe both job requirements and disparate nodes. Further, such an algorithm should find a valid assignment for every job, if such an assignment exists. Also, resources should not be wasted. All other issues being equivalent, a job should not be assigned to a node that is over-provisioned with respect to that job. Finally, the matchmaking process should not add significant overhead to the cost of executing a job. Most of existing approaches for matchmaking in the P2P desktop grid system sacrifice some of these requirements, however, I provide an efficient decentralized matchmaking algorithm that can achieve a good balance among all of these requirements based on a customized Content-Addressable Network (CAN) (as described in Chapter 3).

2. **Comprehensive decentralized load balancing mechanisms**

   Whenever there are multiple candidate nodes in the system that can run a given job (i.e., they can meet the resource requirements of the job), the desktop grid system must consider load balancing among them to obtain high system throughput and low job turnaround times. However, in a decentralized P2P desktop grid system where no centralized information exists, providing good load balancing becomes a challenging problem since available

nodes are heterogeneous and load information propagated in the system can become stale. Even if the initial load balancing mechanism assigned jobs uniformly across available system resources, over time the overall load distribution may change since some nodes can run the allocated jobs much faster than others. Also, whenever jobs arrive at a high rate into the decentralized system, a large number of matchmaking decisions may be made based on stale information, since current load information propagates over time across the nodes in the system. To address these problems, I provide comprehensive load balancing techniques that can initially assign jobs to the available heterogeneous nodes in the system and later redistribute them if needed to improve the overall system throughput (Chapter 4).

3. **A set of optimizations to reduce the system load**

   The load on individual nodes in a desktop grid consists of application load (the jobs to be executed), and system load (load imposed by the workings of the underlying system). Unfortunately, non-uniform distributions of jobs and nodes can cause the system to distribute the system load unevenly across nodes. This system load can come from either monitoring job executions or maintaining the overall P2P system and it can limit the scalability of our system. This overloaded system maintenance cost is not sustainable in the

P2P desktop grid system since every node in our system is a peer. Therefore, unfair distribution of system loads could cause problems to attract the participation of desktop machines into our system. To address these problems, I provide a set of optimizations that can not only reduce the overall system load but also distribute this load more fairly, without impacting the overall reliability or performance of the system (Chapter 5).

1.3 Thesis Organization

The rest of this dissertation is structured as follows. Chapter 2 describes our overall system architecture for executing jobs using a P2P overlay network. Chapter 3 discusses our basic matchmaking framework for any type of resources in the system based on customized Content-Addressable Network (CAN). In Chapter 4, we describe our techniques to improve the overall throughput of our CAN-based system by employing both static and dynamic load balancing schemes. In Chapter 5, we present a set of optimizations to reduce overall system overheads and distribute them fairly among system nodes. In Chapter 6, we perform a large scale experiment to show the ability of our system to scale gracefully as more nodes and jobs are injected. In Chapter 7, we present related work especially focusing on em-

ploying P2P services in Grid computing. Finally Chapter 8 presents conclusions, summarizes the work, and points out possible directions for future work.

Chapter 2

Underlying Framework and Assumptions

In this chapter, we define terminology and the basic framework of our approach for submitting jobs, managing and monitoring jobs while they are running, including methods for failure detection and recovery in a decentralized and distributed environment [56]. Then, we describe our assumed context and overall goals for resource management algorithms.

2.1 Basic Framework

All of the work described assumes an underlying *Distributed Hash Table* (DHT) infrastructure [39, 69, 76, 78, 85, 98]. 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. A system can build upon these basic services to allow users to place idle computational resources into a general pool and draw upon the resources provided by others when needed. 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, similarly to anycasting [73], or content-based routing [2].

A job in our system is 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. 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 *categorical* variables such as operating system type and version. All jobs have modest I/O requirements, with individual input data sets for our initial target applications typically on the order of a few 100 KB or less, with correspondingly small output datasets. However, the jobs for each problem are computationally intensive, since simulation runs consist of advancing physical variables forward in time by solving a set of coupled differential equations, and

data analysis runs perform complex operations on the data. Finally, the jobs in the system are *independent*, which implies that no communication is needed between them (as described in Maheswaran et al. [64]). This is a typical scenario in a desktop grid computing environment, enabling many independent users to submit their jobs to a collection of node resources in the system, or *embarrassingly parallel* workloads. Indeed, Iosup et al. [48] found that a high percent of Grid applications still employ an embarrassingly parallel model based on their analysis on the characteristics of traces of real Grid environments, namely LCG [32] and TeraGrid [90] which are among the largest production Grids currently deployed, and the DAS [31], which is a research Grid.

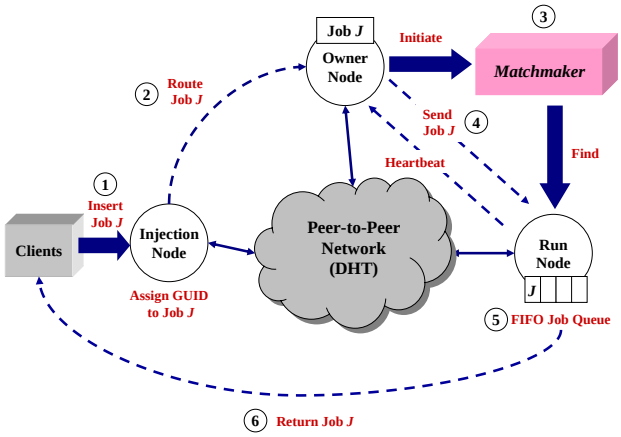


Figure 2.1: Overall System Architecture

Figure 2.1 shows the overall system architecture and flow of job insertion and

execution in the P2P network. The steps of job execution are as follows:

1. A client inserts a job into a node in the system (the *injection node*). The DHT provides an external mechanism that can find an existing node in the system [76, 85].

2. The injection node assigns a Globally Unique IDentifier (GUID) to the job by using its underlying hash function and routes the job to the *owner node*.

3. The owner node initiates a matchmaking mechanism to find a *run node* capable of running the job.

4. Once the matchmaking mechanism finds a run node for the job, the owner node sends the job to the run node.

5. The job is inserted into the job queue of the run node, which processes jobs in FIFO order. While processing the jobs, the run node periodically sends *heartbeat* messages to the owner node.

6. When the job is finished, the run node returns the results to the client.

An owner node is responsible for monitoring the execution of the job and ensuring that its results are returned to the client. Whenever a new job is assigned to an owner node, the owner node attempts to find an appropriate node for running

the job (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 requirements for running the job, such as required CPU speed, amount of memory, supported operating system type(s), etc. Therefore, in the matchmaking process, the first criterion in finding a match is whether the job constraints can be met.

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. Any input data files for a job are transferred to the run node only when the job actually starts running. 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, as the job profile is replicated both on the owner and run nodes to enable reconstruction of job information in case of failures. If either the owner or run nodes fails, the other node will detect the failure and initiate a recovery mechanism to make progress in the job execution. If both the owner and run node fail before the recovery protocol completes, the client must resubmit the job. 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 network routing mechanism.

2.2 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 resource management framework 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, whether 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. The underlying resource management 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 process. 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*. Systems such as Condor [33, 61, 89] mainly target mixed jobs in clustered nodes, while systems like BOINC [3, 4] or SETI@Home [5] deal with clustered jobs in mixed nodes. Our intent is to effectively support all of these

scenarios. To summarize, the goals of any resource management algorithm for a P2P desktop grid system must include the following:

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 evenly distributed across the nodes capable of performing them.

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.

There are additional issues that we do not discuss here. 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 an infrastructure that can accommodate either objective.

Chapter 3

Basic Matchmaking Framework

In this chapter, we describe our basic matchmaking framework based on Content-Addressable Network (CAN) [76] which is customized to allow matching any type of resource requirements of the jobs with available heterogeneous computational resources. First, we describe our basic matchmaking algorithm that can allow *minimum* match for the resource requirements of the job such as minimum required CPU speed or memory amount. Second, we integrate another type of resources into our CAN-based framework which requires a singular value for that resource (i.e., *exact* match) such as a specific type of operating system or processor.

3.1 Basic Matchmaking using CAN

A Content-Addressable Network (CAN) is a DHT that maps GUIDs to points in a d-dimensional space [76] so that nodes divide up the CAN space into (hyper-)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. [87] 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 3.6GHz workstation, with 2GB of memory and 500GB of disk space, to the point $\{360, 2000, 500\}$. A job requiring at least a 1GHz machine, 100MB of memory, and 200MB 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 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*.

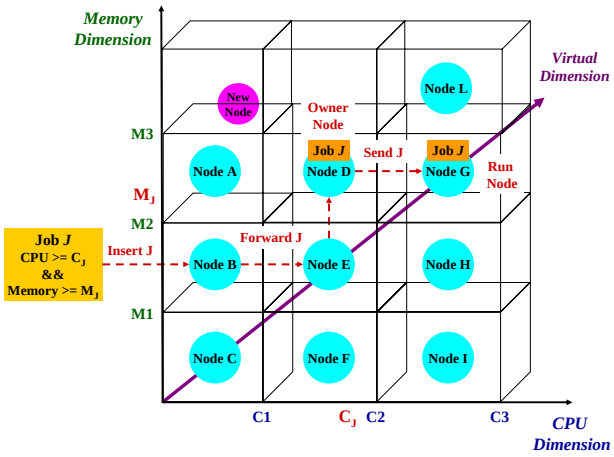


Figure 3.1: Basic Matchmaking Mechanism in CAN

Figure 3.1 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 this dissertation) 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[76]. 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 2.2. 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). *Parsimony* and *Expressive-*

ness 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. 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 Figure 3.1). 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 (represented by 0 coordinates). 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 dimension coordinate. In combination, the randomly assigned node and job coordinates act to break up clusters and spread load more evenly.

3.1.1 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. [76]). 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 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.

3.1.2 Performance Evaluation

In this section, we evaluate our basic CAN-based matchmaking algorithms in decentralized and heterogeneous environments through a comparative analysis of experimental results obtained via simulations. To compare against our CAN-based approach, we evaluate two additional matchmaking algorithms, a *Rendezvous Node Tree*-based approach and a *Centralized Matchmaker*.

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 [85]. 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 deter-

mines 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 GUID in the Chord ring - see details in Kim et al. [51]). 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. [51]). 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 dissertation, the only information distributed through the tree is a description of the maximal amount of each resource available at some node in the subtree. This notion of hierarchical aggregation is a fundamental abstraction for scalability in a large system and is also used in distributed information management systems [77, 95].

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. 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. [51].

Centralized Matchmaker

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. 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 [71, 99].

We can view the Centralized Matchmaker algorithm as the extreme case of the CAN or RNT 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.

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 / τ (τ is the average event inter-arrival time and is set to 0.1 seconds). 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 (as in Zhou et al. [101]).

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, 10000 jobs arrive at the system with an arrival rate of τ). The workloads differ on two axes. Workloads are categorized as either *clustered* or *mixed* (as described in Section 2.2). The former divides all nodes and jobs into a small number of equivalence classes,

where all 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 seconds on 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 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.

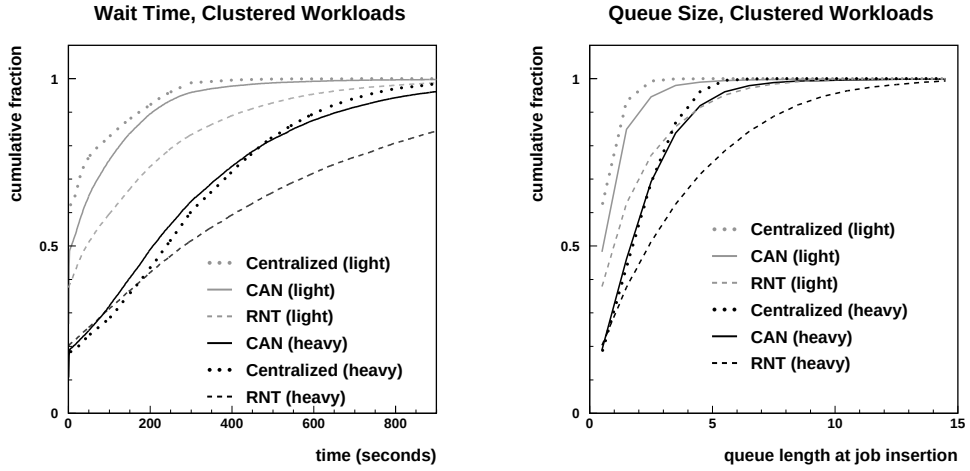
Our metrics are *matchmaking cost* (the number of messages required for finding candidate run nodes by the owner node of a job), *wait time* (the amount of time between when a job is injected and when it actually starts running), and

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 balance. Finally, the distribution of queue lengths provides a direct measurement of the load balance seen by injected jobs.

We test the CAN approach (**CAN**), RNT approach (**RNT**), and the idealized centralized approach (**Centralized**) 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.

Experimental Results

Figure 3.2 and 3.4(a) show wait time, queue length and matchmaking cost (messages) for the clustered workloads, while Figure 3.3 and 3.4(b) show 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

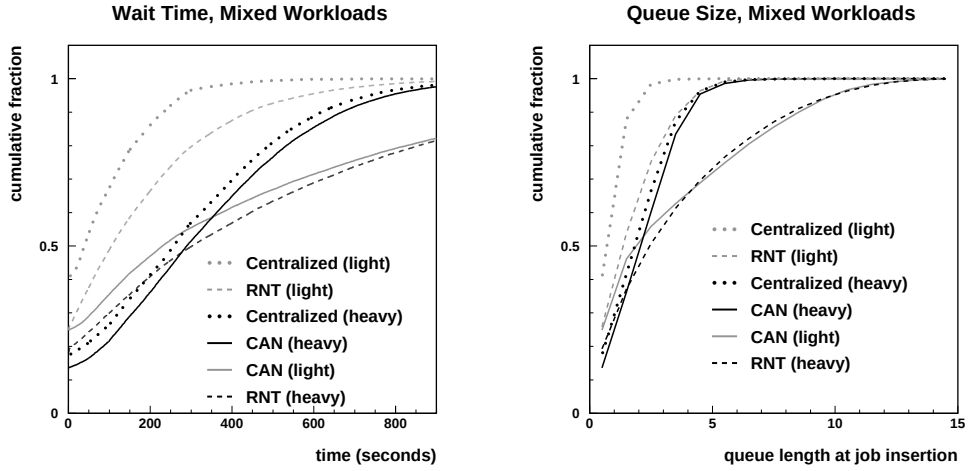


(a) Wait Time

(b) Queue Length

Figure 3.2: Performance Results for Clustered Workloads

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



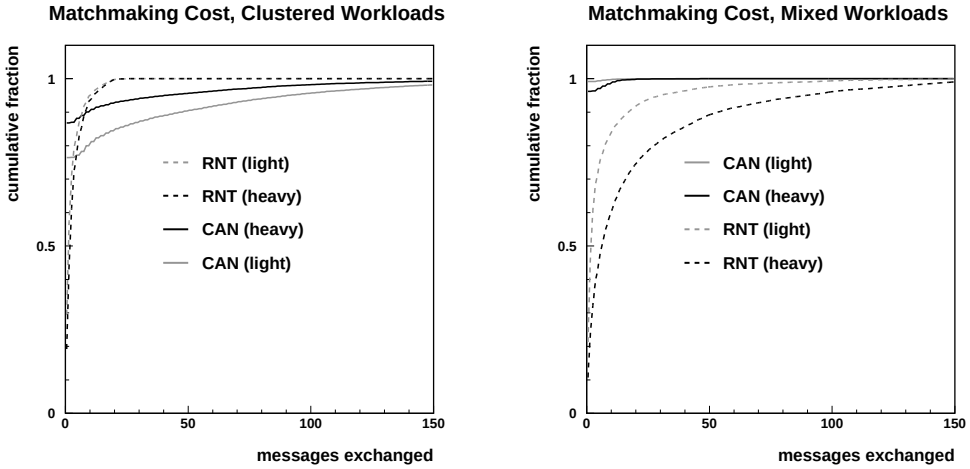
(a) Wait Time

(b) Queue Length

Figure 3.3: Performance Results for Mixed Workloads

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 Figures 3.2(a) and 3.2(b)).

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



(a) Clustered Workloads

(b) Mixed Workloads

Figure 3.4: Overheads of Decentralized Matchmaking

that of RNT. Figure 3.3(b) shows that queue lengths are much larger and more varied in CAN than RNT, implying load imbalance. To understand why the resulting load imbalance 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.

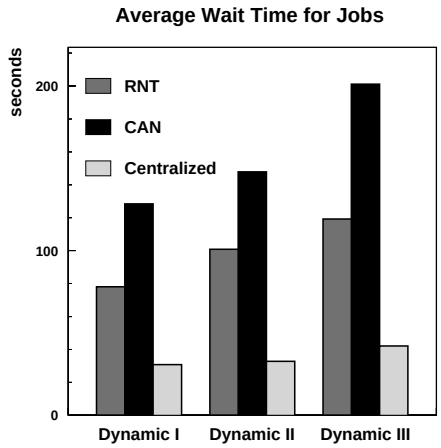


Figure 3.5: Dynamic Workloads

Figure 3.5 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 three dynamic workloads the number of jobs is about 10000,

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, the wait times are worse for CAN than for RNT or Centralized as the overall system becomes more unstable (higher departure rates make the system less stable). 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 re-assigned 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.

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 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 some load information for neighbor nodes. A second potential cause of load imbalance is poor matches between jobs and nodes (i.e, poor

Parsimony). 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 GByte of memory might not improve execution time, and therefore not be useful. Meanwhile, other jobs needing the extra memory might be needlessly queued. By 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.

To summarize, as a comparative analysis on the simulation results shows, we

have identified the benefits and costs of the CAN-based resource management algorithm as follows:

- Overall, the CAN algorithm appears to produce significantly lower wait times than the RNT approach over a broader spectrum of input
- CAN's poor performance with the light mixed workload is an indicative of a broader problem in the robustness of the load balancing
- Reliable job allocations become more crucial to the performance of matchmaking and load balancing in a dynamic environment

To address the load balancing problem of basic CAN-based matchmaking framework with the light and heterogeneous workload, we provide an improved load balancing mechanism based on *pushing* jobs into underloaded regions of the CAN space (as described in Section 4.1). Nodes periodically send load information towards the origin in each dimension. This information is *aggregated* at each step, resulting in each node having partial information about load in all regions of the CAN space containing nodes more capable, which are exactly those nodes that are also able to run that node's jobs. In times of high load, a node can therefore push jobs towards regions of high capability and low load, based completely on local information.

3.2 Categorical Resource Types

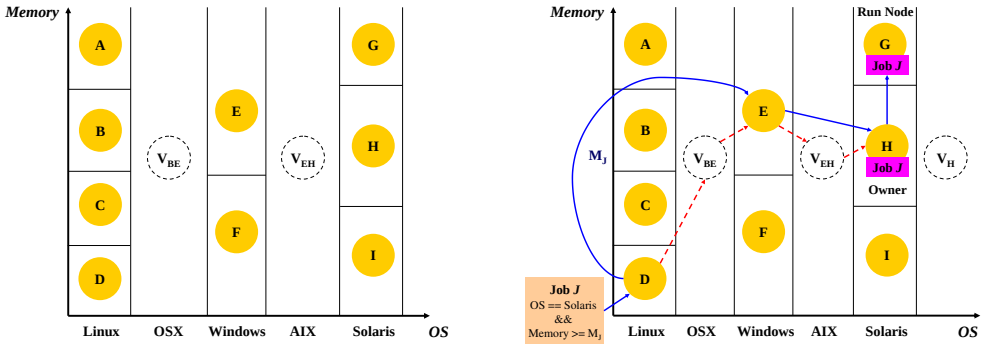
In our system, there are two different types of resources (that can be specified in node capabilities and job requirements): *categorical* and *continuous* resources. Continuous resource constraints such as memory or disk size, or CPU speed require a *minimum* match. On the other hand, categorical constraints require a singular value for that resource (*exact* match), such as a specific type of operating system or processor. Therefore, the system must be able to search for exact matches for the categorical resource types and minimum matches for the continuous resource types simultaneously, while balancing load among multiple candidate nodes.

One example of a possible user query for a set of required resources is (Arch == "Intel" $\wedge$ OS == "Linux" $\wedge$ CPU $\geq$ 2.4GHz $\wedge$ Memory $\geq$ 500MB $\wedge$ Disk $\geq$ 1GB), where Arch and OS are the required processor architecture and operating system type, respectively. To be able to handle this kind of query, the system has to find nodes that both have an Intel architecture and the Linux operating system, and also that meet the remaining continuous resource constraints (i.e., CPU, Memory and Disk).

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

a dimension for architecture and a dimension for operating system in the example). The primary problem with this approach is in specifying the load information that must be aggregated and disseminated throughout the system to perform load balancing (load aggregation mechanism along each dimension presented in Section 4.1). The load information must distinguish between machines with different architectures (e.g., Intel and PowerPC), and also between different operating systems (e.g., Linux and Windows). Moreover, load information must be differentiated on the basis of all *combinations* of these choices; the number of such combinations is exponential in the number of discrete choices for each categorical resource type. A second approach is to create a distinct CAN space for each such combination of choices for categorical resource types. Load information within each such sub-CAN is then homogeneous and can be disseminated efficiently. The drawback of this approach is that such a system requires some type of directory service that vectors incoming jobs to the correct sub-CAN and manages the multiple sub-CANs. This front-end is both a potential performance bottleneck, and also a single point of failure.

Our solution is to integrate categorical resource dimensions into a single CAN space, by **transforming** *them onto a single dimension using a space-filling curve [59, 81]. Then, we address load balancing and connectivity issues by intro-*



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

Figure 3.6: Resource Integration and Routing in a CAN space: In Figure 3.6(b), solid arrows denote the physical routing path of job *J*, while dotted arrows show the logical routing path.

ducing **virtual peers** [57].

Figure 3.6 shows the basic concepts of our approach for integrating categorical resource types into a CAN (as an example, we use operating system for the categorical resource type and memory for the continuous resource type). The basic idea of the approach is to divide the CAN space into multiple *disjoint sub-spaces* where in each sub-space all of the categorical resource types are exactly the same, and provide an efficient mechanism to *connect* the multiple sub-spaces (without having a directory service). For example, in Figure 3.6(a) all nodes in the "Linux" range (A, B, C, and D) have the Linux OS. There is no node that has another oper-

ating system type (such as Windows) in that sub-space. Similarly, nodes G, H, and I have the Solaris OS. The overall CAN space is thus divided into three different sub-spaces, for Linux, Windows and Solaris. The question then is what happens to the rest of the CAN space (i.e., the sub-spaces for OSX and AIX). The OSX and AIX sub-spaces are empty because no nodes have those OS types. Therefore, there can be holes in the CAN space, since a sub-space of the CAN is occupied only if there is at least one real node that has that categorical resource type. However, we cannot just allow holes in the CAN space since they may prevent routing requests from being delivered.

We address this problem by supplementing the "physical" peers with additional *virtual* peers, as shown in Figure 3.6(a), where the OSX and AIX sub-spaces are occupied by two virtual peers V_{BE} and V_{EH}, respectively. Virtual peers act similarly to physical peers, both maintaining neighbor information and allowed to be neighbors of physical peers. However, a virtual peer never is allowed to become a neighbor of another virtual peer, since a single virtual peer can cover multiple unoccupied CAN sub-spaces. Since a virtual peer is not a physical node, we provide a mechanism to map each virtual peer to physical peers (called *manager* nodes). A manager node of a virtual peer maintains all information about the virtual peer (e.g., neighbor list) and processes any routing requests for its assigned

virtual peer(s). In Figure 3.6(a), V_{BE} is managed by nodes B and E while V_{EH} is mapped to nodes E and H. A virtual peer can be managed by up to two different physical peers (the number of mapped physical peers depends on whether the virtual peer is an edge or an internal virtual peer in the integrated CAN space), enabling robust failure recovery.

With this design, each physical peer only is responsible for the exact region of the CAN space to which it belongs, with respect to its categorical resource specifications, and the rest of the space is covered by virtual peers. This enables employing the efficient matchmaking and load balancing techniques presented in Section 4.1, since in each sub-space the existing algorithms can aggregate the load information along the continuous dimensions, and employ the job pushing mechanisms for better load balancing within a single CAN sub-space, without considering different types of categorical resources. Figure 3.6(b) shows the overall procedure of matching a job *J* to node G, showing both physical and virtual peers in the CAN space. Since each virtual peer is mapped to one or two physical peers, a job request can be efficiently delivered to the owner node, as shown in Figure 3.6(b) (e.g., when node D routes the job request to the virtual peer V_{BE}, it can directly send the job to the physical peer E that V_{BE} is mapped to).

3.2.1 1-Dimensional Transformation

In Figure 3.6, we show only a single categorical resource dimension, to simplify the introduction of the concepts of virtual peers, and the description of routing messages across multiple sub-CANs. However, if there are multiple different categorical resource dimensions and we want to divide the CAN space into disjoint sub-spaces, the management of virtual peers and failure recovery mechanisms can become very complex. This is because the number of possible sub-spaces that become empty (so must be covered by virtual peers) increases rapidly with the number of categorical resource dimensions (a combinatorial explosion). The distribution of management of such multiple virtual peers along multiple dimensions and the design of proper failure recovery mechanisms is much more complex than with a single categorical resource dimension.

To address these problems, we *transform* all categorical resource types into a *single* dimension. The overall CAN space is then composed of one transformed categorical resource dimension (we call this dimension T), along with all other continuous resource dimensions (including the virtual dimension described in Section 3.1). Any type of 1-dimensional transformation function can theoretically be used for this purpose, but consider that a user query may specify "don't care" or a limited range query (some of these resource types may also involve

version numbers, which may themselves have ranges) as the requirement for a categorical resource type. In that scenario, the resource query specified for a job becomes a range query in a multi-dimensional space, so that a simple transformation function, such as a row-major or a column-major ordering, results in favoring one dimension over others.

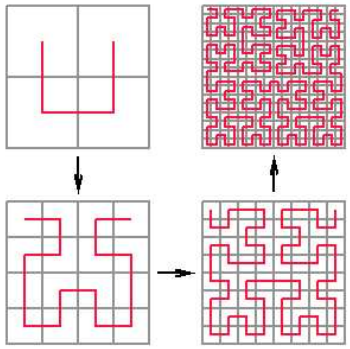


Figure 3.7: Hilbert Space-Filling Curves [72]

Therefore, to transform categorical resource dimensions into a single dimension without favoring any particular resource type, we use a locality-preserving *Space Filling Curve*, specifically the *Hilbert* Space Filling Curve (HSFC) [59, 80, 81]. An HSFC is a continuous mapping from a d-dimensional space to a 1-dimensional space, passing through every point in a d-dimensional space exactly once, resulting in an ordering with good locality properties across all dimensions (as seen from Figure 3.7). Many other research projects have employed HSFCs in

resource discovery scenarios [7, 82].

3.2.2 Virtual Peer Management

Transforming all of the categorical resource types into a single dimension allows us to efficiently introduce virtual peers to cover gaps in the CAN space. If the categorical resources had their own dimensions, then virtual peers would have to cover rectangular holes in the space. In contrast, with a single dimension we can represent a contiguous set of missing configurations (that have no real peer with those values for the categorical dimensions) with a single virtual peer. The result is that for separately managed dimensions, the number of virtual peers is (at worst) *the number of unused configurations*, which grows exponentially as new options are added for each categorical resource type, while for the 1-dimensional transformation the worst case is *the number of **existing** configurations* (plus one).

A virtual peer, like a physical peer, maintains its own neighbor information and periodically updates its information to neighbors. A virtual peer is mapped to physical peers (managers) that perform any CAN-related operations for the virtual peer. So, whenever a virtual peer becomes the neighbor of a physical peer, we add additional mapping information about the virtual neighbor into the neighbor state of the physical peer, to enable efficient routing in the CAN, as shown in

Figure 3.6(b). We now describe the management of virtual peer information when a new node joins the CAN, how job routing works for virtual peers and how to handle failure recovery in the presence of virtual peers.

Node Join Whenever a new node joins the CAN, the representative point for the new node is the combination of the transformed categorical resource dimension (the *T* dimension) coordinates and the other continuous resource type values. The new node splits the zone of one of the existing nodes in the CAN space, specifically the one whose zone already contains the point for the new node. If the new node splits the zone maintained by a physical peer, then the same zone splitting mechanisms used for the *continuous* dimensions is applied, as described in Section 3.1. However, if the new node splits the zone of a virtual peer, this means that the new node is the first physical peer that actually has those values for the categorical resource types. In this case, splitting along *T* dimension, we must split the virtual peer zone correctly.

Figure 3.8 shows the procedure for splitting the zone maintained by a virtual peer upon arrival of a new node *N*. Whenever a new node splits an existing virtual peer's zone, the new node becomes responsible for the newly split virtual peers (i.e., becomes the manager of those virtual peers). In Figure 3.8, there were two

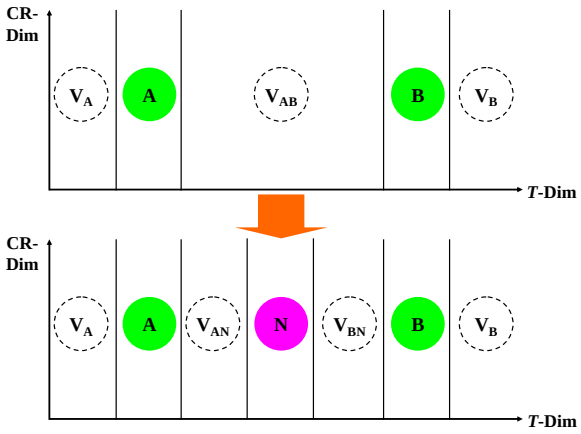


Figure 3.8: Node Join by Splitting a Virtual Peer: T-Dim and CR-Dim denote the transformed dimension and the continuous resource dimension, respectively.

physical peers (A and B) before *N* joins and virtual peers V_A and V_B were managed by node A and node B, respectively. However, V_{AB} was managed by both nodes A and B, because whenever either node A or B departs the system (either a graceful departure or a failure), the other node must be able to recover the lost zone along the *T* dimension. Therefore virtual peer information is replicated in multiple physical peers for failure recovery. Since we transform all categorical resource types into a single dimension, a virtual peer must be replicated in only at most *two* physical peers (a lower and an upper neighbor of the virtual peer along dimension *T*). Suppose new node *N* joins the CAN and splits virtual peer V_{AB}. Since a physical peer maintains only the region it belongs in along the *T*

dimension, the resulting CAN space has four different virtual peers, as shown in Figure 3.8, and node *N* becomes the manager of V_{AN} and V_{BN} that resulted from splitting V_{AB}. Node *N* must then notify physical peers A and B about the changes to the virtual peers that they manage.

There are some special cases where the new node *N* happens to be maintaining a zone *adjacent* to the existing physical peers along the *T* dimension. For example, *N* can join the system by splitting the virtual peer V_{AB} and the zone for *N* becomes adjacent to node A (e.g., the zone maintained by V_{AN} in the Figure 3.8). Then, the node join algorithm should consider this special case where there is no virtual peer between node A and new node *N* (similar for the case where *N* is adjacent to node B). However, we simplify these special cases by allowing *zero-width* virtual peers where even for adjacent nodes there is always a virtual peer between them whose width is actually zero. This makes the node joining algorithm including virtual peer split much simpler and enables our improved job ownership determination mechanism (which are heavily dependent on the virtual peers between sub-CANs) presented in Section 5.2. Therefore, there is only a single and universal case for splitting the virtual peers in the system which is presented in Figure 3.8.

Job Submission Similar to the node join algorithm, whenever a new job is submitted, its categorical resource constraints are transformed into a one-dimensional coordinate and combined with the continuous resource constraints to form a representative point for the job. The job is forwarded to the node that contains the representative point, using the CAN routing mechanism. If the new job ends up at a zone maintained by a physical peer, existing matchmaking and load balancing algorithms are applied within the sub-CAN. Therefore, the overall matchmaking process is to first place a job in the right sub-space where it belongs for its categorical resource constraints, and then do matchmaking and load balancing along the continuous dimensions.

However, if the new job ends up in a zone maintained by a virtual peer, then the system rejects the job since this means that there is no real node that can run the job. (*Completeness*). This is a useful property of the virtual peer design, since physical peers only cover the exact sub-spaces that they belong to, with unoccupied spaces covered by virtual peers.

Failure Recovery An existing node can depart the system at any time either gracefully (meaning the departing node informs its neighbors) or due to failures. The rest of the nodes must then be able to recover the orphaned CAN zone that

was maintained by the departed node. Our failure recovery algorithms are based on the periodic heartbeat messages exchanged between neighbors (as in a standard CAN [76]), with the addition of information related to *sibling* neighbors that abut at the most recent split edge (Section 3.1). Therefore, when the departed node's zone was split along a continuous dimension, that recovery algorithm will be applied to take over the lost zone. If the departed node was one of the managers for a virtual peer, then one of the nodes that ends up taking over the zone becomes responsible for that virtual peer. Since virtual peer information is replicated, the node that takes over the zone can properly initialize the virtual peer information from another live manager. Therefore, as long as there is at least one physical peer in a sub-CAN, the failure recovery algorithm will be applied only along the continuous dimensions.

However, if the departed node's zone was only split along the T dimension (i.e., the departed node was the last one live in that sub-CAN), we reverse the procedure for node join shown in Figure 3.8. Therefore, three sub-spaces are merged into a single zone managed by the virtual peer. Since the virtual peer information is replicated on multiple physical peers, and each node in a CAN space maintains not just its neighbor information, but also neighbor of neighbor information [76], the algorithm can recover the zone and initialize the appropriate

virtual and physical peers. By transforming all categorical resource types into the one-dimensional space, we obtain the benefits of a straightforward, robust failure recovery algorithm that only has to work differently from the earlier algorithm for the single transformed dimension.

3.2.3 Scalability Issues

Although our proposed design for integrating all types of resources based on virtual peers and 1-dimensional transformation can effectively match incoming jobs with various types of resource constraints to available heterogeneous resources, it has drawbacks in terms of system maintenance.

As shown in Figure 3.6, a virtual peer becomes the neighbor of *all* physical peers that abut in the *T* dimension (for example, V_{BE} is the neighbor of nodes A, B, C, D, E, and F). This means that a virtual peer must exchange heartbeat messages with many neighbors periodically, and the size of each message grows with the number of the virtual peer's neighbors (this is because each node in CAN sends its own information and its neighbor information in a periodic update message [76]). Therefore, such messages can add substantial overhead for the nodes responsible for the virtual peers. For example, if a virtual peer manages 1000 neighbors then the size of a single heartbeat message is about 600KB and this can

become a significant burden since a virtual peer sends update messages to *all* its neighbors (i.e., the total size of the messages would be 600MB at every update).

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

Another potential problem with the virtual peers is that some nodes may process more routing messages than other nodes. As described in Section 2.1, routing a submitted job starts from an injection node, and we assume that this injection node is chosen randomly from the available nodes. Therefore, some jobs start from the desired sub-space where the categorical resource types are already met. Other jobs however may start from a completely different sub-CAN. For example, in Figure 3.6, a job that requires the Linux operating system type may start from

node H (which is the injection node of this job), where only Solaris machines are located. This means that some jobs must traverse multiple sub-spaces until they arrive at the right sub-CAN (in terms of categorical resource types). In this step of matchmaking, the manager nodes may be heavily used for routing jobs, since to move from one sub-space to another sub-space jobs must traverse the virtual peers. Therefore, the manager nodes can suffer from processing a large number of routing messages.

All of these issues can limit system scalability as they complicate system maintenance with increasing numbers of nodes and jobs. We address the problem of heartbeat message exchanges between virtual peers and physical peers by employing *modified* heartbeat messaging scheme in Section 5.1. This is one of our efforts to minimize any overheads in the system and distribute them fairly among system nodes to build an effective and scalable P2P desktop grid. In the following section, we discuss our technique to balance the overhead of processing routing requests across multiple sub-CANs by using *specialized* routing in the T dimension.

Specialized Routing in the T Dimension

Since a job can be injected at any node in the CAN space, it may have to traverse multiple sub-spaces to reach the sub-space where it belongs, to match its categorical resource requirements. Due to this property of job routing, the manager nodes for virtual peers can be a bottleneck in processing routing messages. We address this problem by using a special routing mechanism in the *T* dimension.

Whenever a physical peer tries to route a request to the virtual peer, it sends the request to one of the *neighbors* of the virtual peer (rather than sending directly to the manager of the virtual peer). Therefore, in the *T* dimension, the algorithm utilizes the neighbor of neighbor information maintained by the CAN, and routing requests are processed not only through direct neighbors but also *indirect* neighbors. For example, in Figure 3.6(b), when node D routes the request for job *J*, it selects one of V_{BE}'s neighbors (nodes E or F) and sends the request. This prevents all routing requests delivered from the Linux sub-space to another sub-space from always going through node E, the manager of V_{BE} [57].

3.3 Summary

In this chapter, we have presented our basic matchmaking framework for matching minimum and exact resource requirements specified by the jobs with available heterogeneous computational resources.

First of all, to handle continuous resource types which require minimum matches for those resources, we modified the Content-Addressable Network (CAN) [76] by treating each resource type as a distinct dimension in the CAN space. However, the basic CAN procedure encounters difficulties when many nodes have similar, or even identical, resource capabilities. Since the coordinates of a node are defined by its resource capabilities, identical nodes are mapped to the same point in the CAN volume. This creates a problem for the one-to-one mapping of nodes to zones. Additionally, many jobs might have very similar requirements so that they can be mapped to a comparatively small region of the CAN space which results in load imbalance. We addressed this problem by augmenting both job and node descriptions with a randomly assigned value in a virtual dimension. The virtual dimension ensures that all jobs and nodes are unique, and helps balance load even when the actual jobs and nodes are similar. The resulting system can achieve a good balance among multiple goals for matchmaking frameworks presented in Section 2.2. However, we found that the CAN-based algorithm works very poorly

due to serious load imbalance when jobs with few requirements are run on nodes with heterogeneous (mixed) resource capabilities. We will address this problem by employing advanced load balancing mechanism in Section 4.1.

Second, to integrate categorical resource types which require exact matches into a single CAN space, we transformed them onto a single dimension using a space-filling curve. Then, we addressed load balancing and connectivity issues by introducing virtual peers. With this design, each physical peer only is responsible for the exact region of the CAN space to which it belongs, with respect to its categorical resource specifications, and the rest of the space is covered by virtual peers. This enables employing the efficient matchmaking and load balancing techniques, since in each sub-space the load information is homogeneous and can be disseminated efficiently, without considering different types of categorical resources. However, as we discussed in Section 3.2.3, this can distribute the system load unevenly across nodes due to highly non-uniform distributions of jobs and nodes in the system. We will address this problem by providing a set of optimizations that can improve the scalability of our system in Section 5.1.

Chapter 4

Load balancing of Job Executions

In this chapter, we present our comprehensive load balancing techniques which employ initial static load balancing mechanism and supplemental dynamic load balancing scheme to improve overall system throughput and user response time. First, we address the load imbalance problem discussed in Section 3.1 by employing pushing mechanism based on dynamic aggregated load information. However, these improvements are still based on a static allocation of jobs to nodes in the system which can cause potential load imbalance due to the heterogeneity of nodes and jobs and stale load information. Therefore, we supplement our initial and static load balancing scheme by using lightweight and effective dynamic load balancing mechanisms.

4.1 Improved Static Load Balancing

In previous chapter, we showed that the CAN-based matchmaking mechanism can achieve good load balancing among the multiple candidate run nodes with low matchmaking cost in most scenarios. However, we found that the CAN-based algorithm works very poorly due to serious load imbalance when jobs with few requirements are run on nodes with heterogeneous (mixed) resource capabilities (as we discussed in Section 3.1).

We now describe how we have improved the basic CAN-based matchmaking mechanism to address this problem by *pushing* jobs into underloaded regions of the CAN space based on *dynamic aggregated load information* [52, 53].

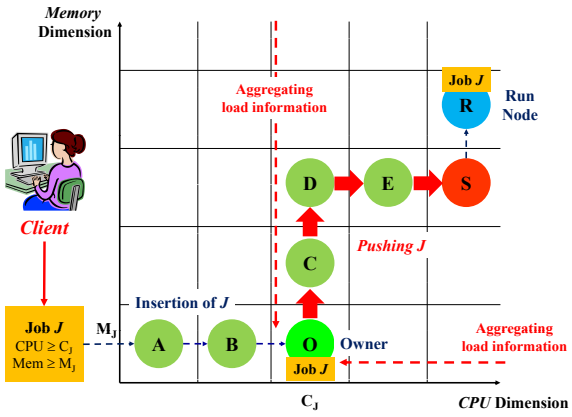


Figure 4.1: Improved Static Load Balancing Mechanism

Figure 4.1 shows the basic concepts of our improvements. When a new job is

inserted into the system and routed to the owner node (node O), the job is *pushed* into an underloaded region in the CAN space. To determine whether to initiate pushing of a job, a fixed amount of current system load information is propagated along each dimension in the CAN space. If the overall system is lightly loaded, the job can be pushed into the upper regions of the CAN space (farther from the origin) and utilize the more capable nodes in the system (node R). We cannot push jobs to lower regions (closer to the origin) in the CAN space, because the nodes occupying those regions will likely not be able to satisfy the jobs' requirements. It is very important that each node in the pushing path of a job be able to make the decision whether to continue pushing the job in a completely decentralized fashion, based only on local information. Therefore, the amount of information maintained by each node for pushing jobs should remain constant with respect to the number of jobs.

4.1.1 Enhanced CAN Mechanism Details

To enable the pushing of a job to an underloaded region in the CAN, we have to propagate a fixed amount of current load information through the nodes in the CAN space. Since each node cannot maintain an accurate global picture of the system load, the load information must be properly *aggregated*. Also, the load

information should be *dynamic* so that it can reflect the current distributed state of the system. For this dynamic aggregated load information we use the following measures along each dimension in a CAN space:

- *Number of Nodes*
- *Sum of the Job Queue Sizes*

We add this aggregated load information to the periodical neighbor state update mechanism of the original CAN DHT maintenance algorithm [76], to avoid generating additional messages in the P2P network. By using the two aggregated load statistics, for a given node *N* we can estimate the current load (e.g., average job queue size) along each dimension of the CAN for the nodes that own CAN regions with greater values than that of node *N* in that dimension. However, it is not easy to accurately compute the aggregated load information, since the overall CAN space can be *irregularly* partitioned. To build a regularly partitioned CAN space, the representative points for all nodes in the system should be distributed uniformly. In our CAN, the point for a node consists of its resource capabilities and an additional virtual dimension coordinate. Therefore we cannot assume that the resource capabilities of the nodes in the system have a uniform distribution since, in the real system, only a small portion of the nodes are likely to have

high resource capabilities, with the majority of the nodes having relatively lower capabilities [101].

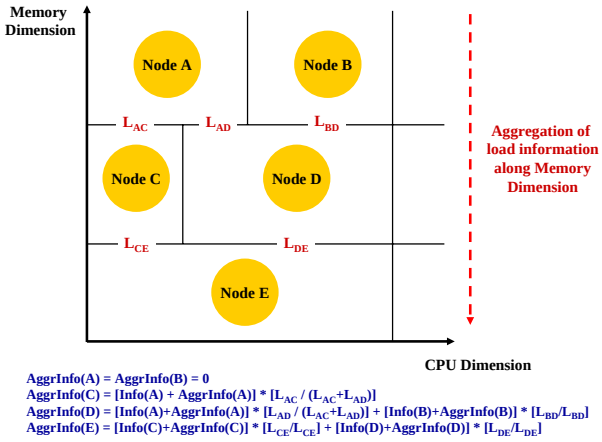


Figure 4.2: Computing Aggregated Load Information

To deal with aggregation of load information in the irregular CAN space, the algorithm uses an *overlap fraction*-based computation, as shown in Figure 4.2. Figure 4.2 shows the process for aggregating load information along the Memory dimension in a CAN space. Info(N) is the current load information for node N (e.g., job queue size on N). AggrInfo(N) is the computed aggregated load information from nodes with Memory values greater than that of node N (`Number of Nodes` or `Sum of the Job Queue Sizes`). Whenever a node N computes its aggregated load information, it only carries some fraction of the information from its neighbors with larger Memory values, depending on how much

N's boundary overlaps with those neighbors. Note that the information about the neighbors is propagated through the periodical CAN neighbor state update mechanism. More generally, for each dimension *d* in a CAN space, node *N* can compute the aggregated load information along the dimension *d* (denoted by $AI_d(N)$) as follows:

$$AI_d(N) = \sum_{u \in UN_d} (AI_d(u) + I(u)) \times OF_d(N, u) \tag{4.1}$$

$$OF_d(N, u) = \frac{\prod_{i \neq d} OverlapEdge(u, N, i)}{\prod_{i \neq d} Edge(u, i)} \tag{4.2}$$

In Equation 4.1, UN_d is the set of nodes adjacent to N with which it shares a border along N's upper edge in dimension d. For Node D in Figure 4.2, and considering the memory dimension, this would be the set {Node A, Node B}. For each node u in UN_d, N adds the local and aggregated information from u and multiplies it by a factor $OF_d(N, u)$. This factor reflects the fact that nodes other than N might have u as a neighbor in dimension d (for example, Node C also has Node A as a neighbor), so without the multiplier u's information will be included more than once (when Node E aggregates information from both Node C and Node D). In particular, if LN_d are the lower neighbors of u at dimension d

(thus $N \in LN_d$), then it must hold that

$$\sum_{v \in LN_d} OF_d(v, u) = 1$$

in order for u's load information to be aggregated in full along dimension d (Node A's information must be split between Node C and Node D).

The aggregation multiplier $OF_d(N, u)$ is the overlap fraction of N and u along dimension d, from the perspective of node u. That is, if N and u control adjacent hyper-volumes in the CAN space, it is the fraction of u's hyper-area at its lower bound in dimension d that intersects with N's hyper-area at its upper bound in d. In two dimensions, it is the length of the line segment describing N and u's shared border divided by the full length of u's bordering edge. For example, $OF_{memory}(D, A) = L_{AD}/(L_{AC} + L_{AD})$, where L is the length of the line segment. In higher dimensions, the orthogonality of the dimensions means that we can compute each of these linear fractions for the dimensions other than d, and take their product to obtain the overlap fraction. This is what is shown in Equation 4.2, where $OverlapEdge(u, N, i)$ is the overlap of u and N in dimension i (L_{AD} for Node D and Node A in the CPU dimension) and $Edge(u, i)$ is the length of u's edge in dimension i ($L_{AC} + L_{AD}$ for Node A in the CPU dimension).

Once the aggregated load information is propagated through the entire CAN space, all the way to the nodes near the origin, the system is able to push the

incoming jobs into underloaded regions for better load balancing and to utilize more capable nodes in the system. To initiate the job pushing we have to address several issues as follows:

1. *Target Node* - Where should a job be sent?

2. *Stopping Criteria* - When should pushing be stopped?

3. *Criteria for the Best Run Node* - Which candidate run node should be selected?

To determine the target node, first we want to push the jobs into lightly loaded regions of the CAN space. Likely the best way to determine the load of the system is to use the aggregated average job queue size. Since each node has aggregated load information about each upper neighbor locally, it can calculate the aggregated average job queue size for each upper neighbor by using `Number of Nodes` and `Sum of the Job Queue Sizes` carried by the load propagation mechanism. However, the shortest average job queue size does not always give the best choice. A node with a slightly longer aggregated average queue size might also enable access to a larger number of potential run nodes than the node with the smallest aggregated average queue size. This 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, we want to push jobs to the upper neighbor node that has both a small aggregated load (average job queue size) and a large number of available nodes above that neighbor node, to increase the number of candidate run nodes. To summarize, we can determine the target node based on the following objective function:

$$F_d(u) = \frac{AI_d(u).SumOfJobQueueSizes}{(AI_d(u).NumberOfNodes)^2} \tag{4.3}$$

Whenever a node chooses a target node from among its upper neighbors, it calculates $F_d(u)$ for each $u \in UN_d$ and picks the one that has the minimum objective function value across all dimensions.

By using the objective function in Equation 4.3, each node in the path of a pushed job can decide where to push the job based only on local information. The question then is the stopping criteria – *when* should pushing be stopped? We must avoid pushing jobs to the extreme edges of the CAN space, because that will result in load imbalance. The stopping criteria for pushing a job should reflect the current (but distributed) load of the system and be computed based only on each node's local information. The very first condition for stopping should be whenever the matchmaking mechanism finds a *free* node that meets the resource requirements of a job; then matchmaking can stop pushing the job and assign the

free node as the run node. Note that each node can determine whether there is a free node in its neighborhood based only on its local neighbor state information, which is updated periodically. In a relatively lightly loaded system, this mechanism works well, since every time the matchmaking is performed, it can find a free node in the system. However, in a heavily loaded system where most, if not all, of the nodes are already busy processing jobs, it is not clear how we stop pushing a job without causing severe load imbalance. A simple way to do this is for each node to estimate the current load (average job queue size) of its surrounding neighbors, and if the load is below a predefined threshold, then it can stop pushing and assign the job to one of its neighbor nodes. However, to determine a threshold that is insensitive to the characteristics of various workloads is not trivial. Therefore, we employ *probabilistic stopping* according to the following formula:

$$PS(N) = \frac{1}{(1 + AI_{TD}(N).NumberOfNodes)^{SF}} \tag{4.4}$$

In Equation 4.4, $PS(N)$ shows the probability to stop pushing a job from node *N*, and SF is the *stopping factor*, which greatly affects the shape of the probability function. As the number of nodes above node *N* in the target dimension TD (determined by the neighbor minimizing Equation 4.3) becomes smaller, the prob-

ability of stopping becomes greater. This means that if a job approaches the edges of the CAN space, with high probability the pushing will stop and a run node is chosen based on local information. This feature avoids pushing incoming jobs to the edges of the CAN space, which would overload the nodes near the edges. We can adjust the probability function by changing SF (higher SF means a higher probability of pushing the job). We tested three different SF values from 1 to 3 and show the experimental results in Section 4.1.2.

We have shown (1) how to aggregate the dynamic load information in a CAN space (Equations 4.1 and 4.2), (2) based on that information how to choose a target node for a job (Equation 4.3), and (3) when to stop pushing a job (Equation 4.4). The final step in the matchmaking algorithm is to choose the *best* run node among the multiple candidates. Pushing of incoming jobs can be stopped either because the matchmaking mechanism found a free node or due to the probabilistic stopping function. In the former case, the node where the pushing stopped (we call this node the *matching node*) creates a list of capable candidates using its local neighbor state information. It is possible that there might be multiple free nodes among the candidates, in which case the matchmaking algorithm selects the *fastest* candidate run node (measuring CPU speed), since that can speed up the overall processing of a job. However, if the pushing process stopped because

of the probabilistic stopping function, this means that there are not enough free nodes in the system. To choose the best run node from among the candidates, but with no available free nodes, we use the following score function for ranking the candidates:

$$F(C) = \frac{C.JobQueueSize}{C.SpeedOfCPU} \tag{4.5}$$

In Equation 4.5, $F(C)$ is the score function for a candidate run node C. The candidate node with the minimum score will be selected as the best run node: the algorithm prefers a node with a smaller job queue and a faster CPU. Using only the set of candidate run nodes built by the matching node may not be sufficient, since we are pushing the jobs across multiple nodes in the system. Therefore, we still consider the candidate run nodes found in the process of pushing, in addition to the candidate run nodes around the matching node, for better load balancing. To summarize, at each step of pushing a job, the matchmaking mechanism keeps the best candidate run node based on the score function in Equation 4.5, and considers it in the list of candidates created by the matching node whenever the matchmaking mechanism cannot find a free node in the system.

4.1.2 Performance Evaluation

In this section, we evaluate our improved load balancing algorithms in decentralized and heterogeneous environments and present a comparative analysis of experimental results obtained via simulations. To compare against our CAN-based approach, we evaluate two additional matchmaking algorithms, a Rendezvous Node Tree-based approach and a Centralized Matchmaker that were described in detail in Section 3.1.2.

Experimental Setup

To see the behaviors of our system with improved static load balancing techniques, we use our event-driven simulator described in Section 3.1.2. Events include node joins, node departures (graceful or from a failure), and job submissions. The events are generated using a Poisson distribution with an arrival rate of $1/\tau$ (τ is the average event inter-arrival time). 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 as described in Section 3.1.2.

Our test traffic workloads differ on two axes. Workloads are categorized as either *clustered* or *mixed* (as described in Section 2.2). The former divides all

nodes and jobs into a small number of equivalence classes, where all 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 are "lightly" or "heavily" constrained. For a given job, each type of resource has a fixed independent probability of being constrained: "lightly-constrained" jobs have an average of 1.3 constraints (out of the 3) and "heavily-constrained" jobs have an average of 2.4. As a job has more minimum resource requirements (heavily-constrained workloads), it is likely to be harder to match the job since fewer nodes in the system can meet those multiple constraints. We present only results from *mixed* workloads since in the clustered workloads, the CAN-based matchmaking mechanism already has shown better performance than the RNT based approach and is close to that of the Centralized Matchmaker (Section 3.1.2).

The amount of work W for a job j is generated uniformly at random from a predefined set of work ranges (40 minutes on average), and means that to run the job j a node must execute for W time units if it has exactly the same node specification as does the job j's constraints. To model the actual running time of a job, we divide W by the node CPU speed (relative to some baseline node CPU speed), to get a run time on the node a job is assigned to. Finally, for the network communication cost, the average latency of a packet between any two nodes in

the system is set as 50 milliseconds which is exponentially distributed.

Our metrics are *matchmaking cost* (the amount of time between when a job is injected and when it is assigned to a run node in the system), *wait time* (the amount of time between when a job is injected and when it actually starts running) and *average queue length* (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 overhead 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 of a run node before a job is executed. Wait time reflects both protocol overhead and the quality of the matchmaking results, i.e., load balancing. Finally, the distribution of queue lengths provides a direct measurement of the load balance seen by injected jobs.

We test the original CAN approach (Section 3.1) (**CAN**) and the improved CAN approach employing dynamic aggregated load information with different stopping factors from 1 to 3 (**CAN-P1,2,3**). To compare against CAN-based matchmaking mechanisms, we also tested the RNT based approach (**RNT**) and the idealized centralized approach (**CENTRAL**). We do not include matchmaking cost for the centralized approach because it incurs no cost for matchmaking.

Performance Results

We begin by discussing the experimental results obtained from relatively static workloads with lightly and heavily-constrained jobs, respectively. In the static workloads, no nodes join or leave the system during the course of the experiments. There are six different workloads for the lightly-constrained jobs, which have different values of τ (average inter-arrival time of jobs) from 15 seconds to 20 seconds. Similarly, for the heavily-constrained workloads, we varied τ from 25 seconds to 30 seconds.

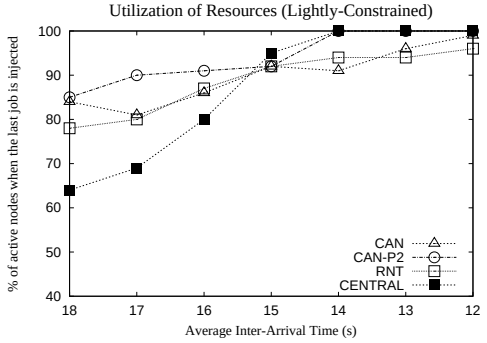


Figure 4.3: Utilization of Resources for Lightly-Constrained Workloads

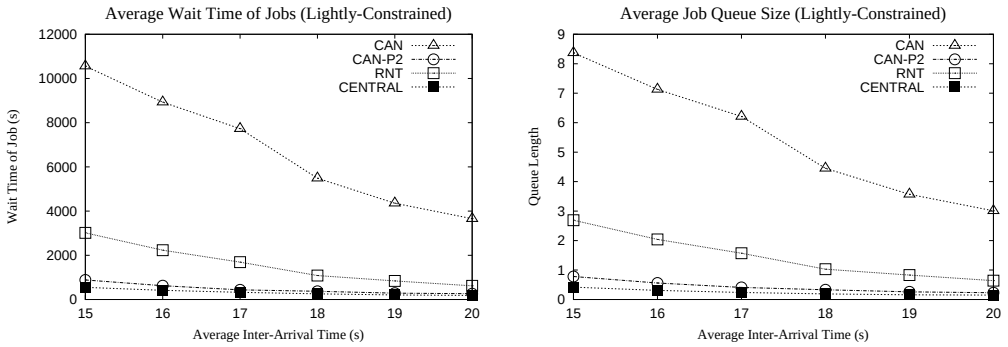
The important characteristic of these workloads is that all of them reach a *steady state* during the simulation period. For example, the percentage of active nodes (nodes currently running jobs) when the last job is injected into the system for lightly-constrained workloads is depicted in Figure 4.3. Figure 4.3 shows

that for values of τ from 18 down to 16 seconds, the utilization of the overall system resources remains low, indicating lightly loaded environments, while from 14 seconds down almost 100% of the nodes are busy processing other jobs when the last job is inserted into the system. This means the system has reached its maximum throughput. Interestingly, the utilization of CENTRAL is smaller than all other matchmaking mechanisms in lightly loaded environments (from 18 to 16 seconds). This is because CENTRAL is the global algorithm that can assign a job to the fastest idle node in the system, which accelerates the rate at which jobs are processed.

In the steady state, the rate for incoming jobs and finishing jobs is approximately the same, and we want to show the performance of each matchmaking mechanism in this steady state, to avoid the transient effects of earlier jobs that see a largely empty system. We can inject more jobs with smaller τ to increase the system load, which will eventually saturate the system and result in indefinite growth of job queues. However, this will not be feasible in a real system, since when the overall system becomes too heavily loaded the system can refuse to receive more jobs until it becomes stabilized.

The desire to measure steady state behavior explains why we choose different ranges for τ for lightly and heavily-constrained jobs. In the heavily-constrained

workloads, many jobs have multiple resource requirements, and this reduces the number of nodes that are legal matches for a job in the system. Therefore to make the workloads reach steady states, we increase τ for these jobs relative to the lightly-constrained workloads. The workloads belonging to either the lightly or heavily-constrained sets have exactly the same job and node profiles, respectively, so that we can directly compare across different values of τ.

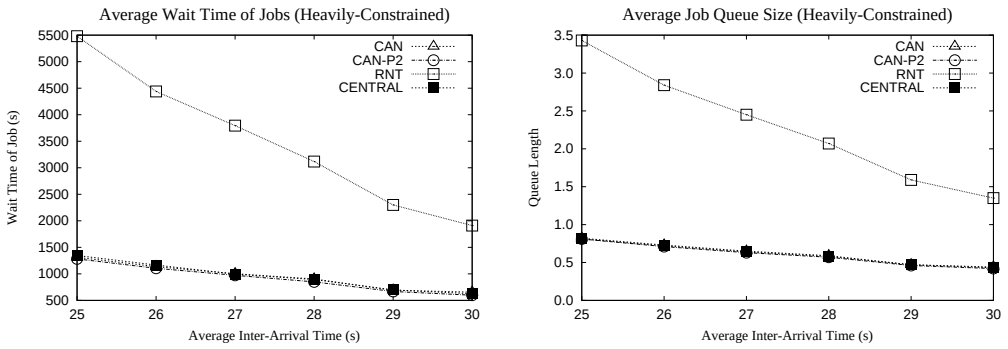


(a) Average Job Wait Time

(b) Average Job Queue Length

Figure 4.4: Performance Results for Lightly-Constrained Workloads

Figures 4.4(a) and 4.4(b) show the performance results for the matchmaking mechanisms, measuring job wait time and queue length for lightly-constrained workloads. We only plot the improved CAN-based matchmaking mechanism with stopping factor 2 (CAN-P2) since it shows relatively stable performance for both lightly and heavily-constrained workloads (insensitive to the characteristics

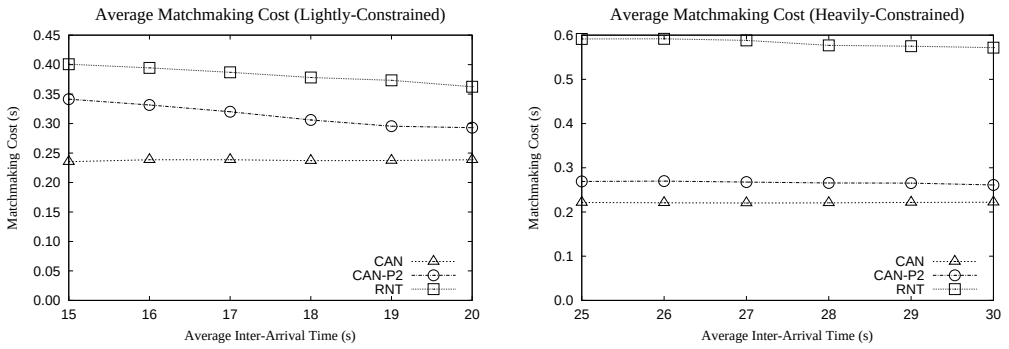


(a) Average Job Wait Time

(b) Average Job Queue Length

Figure 4.5: Performance Results for Heavily-Constrained Workloads

of the workloads). The results imply that our improved CAN-based matchmaking mechanism shows very competitive performance even compared to CENTRAL and improves the quality of load balancing dramatically from the original CAN algorithm (CAN). More specifically, CAN-P1 has 2.1 times the average job wait time of CENTRAL across all the lightly-constrained workloads, CAN-P2 is a factor of 1.5 worse and CAN-P3 is a factor of 1.4 worse, while the RNT is a factor of 4.6 worse and CAN 21.2 times worse. The main reason CAN has poor load balancing is that for the lightly-constrained workloads, a majority of the jobs have few or no constraints, so that many jobs are mapped to a comparatively small region of the CAN space near the origin. More specifically, if a job does not specify any requirement for a specific resource type, the corresponding coordinate for the



(a) Lightly-Constrained (b) Heavily-Constrained

Figure 4.6: Average Matchmaking Costs

job is mapped to the minimum constraint value (in our case, 0), and this results in a *hot spot* causing load imbalance. However, by pushing jobs to underloaded regions of the CAN space, CAN-P2 can disperse the jobs in the different dimensions from the original hot spot, which results in superior load balancing (as seen in Figure 4.4(b)). Additionally, CAN-P2 can utilize more capable nodes whenever needed, which can accelerate overall job processing so that CAN-P2 also outperforms the RNT.

However, pushing jobs in the CAN space may cause additional overhead for matchmaking, since each job must traverse the CAN space from its owner node to find an appropriate run node. Figure 4.6(a) shows that CAN-P2 has worse matchmaking performance than CAN. Also, as we increase the stopping factor (SF), the

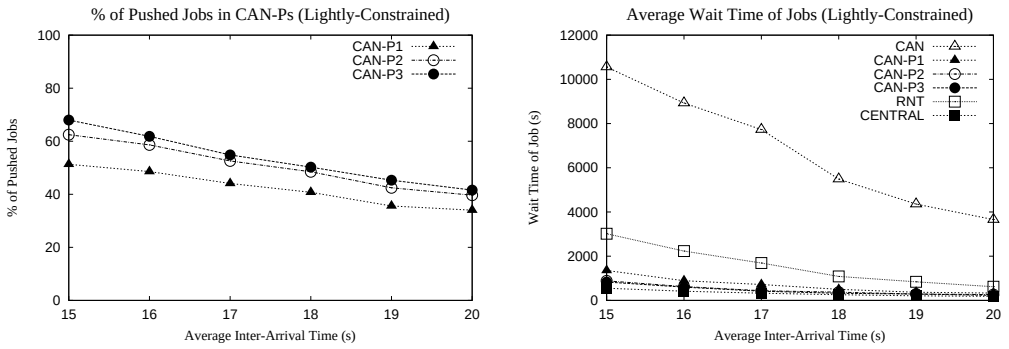
matchmaking cost increases accordingly, since with higher SF the probability for stopping decreases. However, all of the CAN-based matchmaking mechanisms (CAN and CAN-P2) still show better matchmaking performance than RNT. This is because the CAN-based matchmaking mechanism inserts each job into the right place in the DHT for matchmaking (the owner node), where surrounding neighbor nodes can already meet the resource requirements of the job. However, in the RNT approach each job starts from a completely random place in the DHT and must find an appropriate run node for the job through searching up and down the RNT. Another interesting result in Figure 4.6(a) is that all of our matchmaking algorithms (including CAN, CAN-P2 and RNT) show very low cost for performing matchmaking in distributed and heterogeneous environments. Compared to the wait time of jobs shown in Figure 4.4(a), the cost for matchmaking is almost negligible. This could be because of our assumption about the average packet delay for a message, which is set to 50 milliseconds. However, even considering this packet delay, the results show that all of our matchmaking mechanisms find an appropriate run node with a very small number of P2P network hops to achieve good load balancing. Hence, we can concentrate on the load balancing issue whenever the average running time of jobs (in our case, 40 minutes) is significantly longer than the network communication speed, which is a typical scenario in a desktop

grid computing environment.

Heavily-Constrained Workloads The results in Figure 4.5 show quite different stories about the performance of matchmaking algorithms. Figure 4.5(a) and 4.5(b) show that all of the CAN-based matchmaking mechanisms obtain performance very close to that of CENTRAL when measuring load balance, while RNT shows the worst performance among all the matchmaking mechanisms. For the heavily-constrained workloads, many jobs have multiple resource constraints, which limits the number of nodes in the system that can be matched to a job, so that the CAN-based mechanisms can achieve very good load balancing even compared to CENTRAL [54].

Although we cannot directly compare the results in Figure 4.6(b) with Figure 4.6(a), the gap between the RNT and CAN-based mechanisms appears larger for the heavily-constrained workloads. This is because the RNT search suffers heavily from trying to find appropriate run nodes for jobs with multiple resource requirements.

Costs and Benefits of SF Different stopping factor values can affect the behavior of the CAN-P algorithm, measuring the number of jobs pushed, as seen in Figure 4.7(a). With higher SF, more jobs will be pushed into the upper regions of



(a) % of Pushed Jobs

(b) Average Job Wait Time

Figure 4.7: Costs and Benefits of CAN-P for Lightly-Constrained Workloads

the CAN space due to the decreased stopping probability, so that CAN-P3 shows the highest percentage of pushed jobs among the three different CAN-Ps. Increasing the stopping factor increases the overall matchmaking cost, since jobs are pushed farther in the CAN space to find appropriate run nodes. However, that does provide benefits from better load balancing, as seen in Figure 4.7(b), since more capable nodes end up being used for some jobs in the system. As the overall system becomes lightly loaded (increasing τ), the percentage of pushed jobs decreases, since the matchmaking mechanism is more likely to encounter an empty node (as seen from Figure 4.7(a)). The decrease is less for heavily-constrained workloads since there are not as many nodes in the system that can run the incoming jobs, which means that the jobs start pushing from relatively near the edges of

the CAN space.

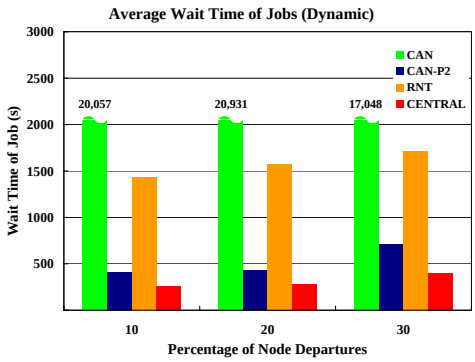


Figure 4.8: Experimental Results for Lightly-Constrained Dynamic Workloads

Dynamic Workloads Figure 4.8 shows wait times for three *lightly-constrained mixed* workloads, where between 10% and 30% of the nodes leave during the course of simulation, and shows that node departures can affect CAN-P's ability to match CENTRAL's performance. The value of τ for all of the dynamic workloads is set at 17.5 seconds. Note that in Figure 4.8, results from the basic CAN are truncated since they have very large values compared to the other matchmaking frameworks. Node departures include graceful departures, where a node informs its neighbors before leaving, and failures, where the neighbors learn of the departure from missing P2P network heartbeat messages. All of the dynamic workloads have the same number of jobs and the same job profiles, but have dif-

ferent sets of available nodes in the system at different times, so that we cannot directly compare across workloads.

In the dynamic workloads, because existing nodes depart the system the information carried by the CAN- and RNT-based mechanisms can be more stale compared to the information maintained for static workloads, and there can also be some overhead for P2P network recovery (unlike for CENTRAL). More specifically, CAN-P2 shows 1.6 times the job wait time of CENTRAL on average across all the workloads, and RNT is a factor of 5.2 worse. Although we cannot directly compare these results with Figure 4.4, clearly there are some load balancing issues for both the CAN-P and RNT algorithms, that keep them from approaching the wait time performance of CENTRAL. The dynamic behavior of the nodes in the system seems to have a much larger impact on basic CAN compared to CAN-P2 or RNT. Since all of the dynamic workloads are based on mixed sets of nodes and jobs, a load imbalance problem similar to the one that we saw for the basic CAN earlier, due to a hot spot in the CAN space, can occur as the 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 re-assigned to other live nodes in the system. Since each node in the hot spot

has a disproportionate number of assigned jobs, this causes even more severe load imbalances. However, by employing the pushing mechanism based on dynamic aggregated load information, CAN-P2 can spread the jobs away from the hot spot and achieve more reliable load balancing compared to CAN and still outperforms the RNT, which is based on random initial load balancing.

4.2 Dynamic Load Balancing

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

Static load balancing has drawbacks, both because of *heterogeneity* in the running times of jobs and in the resource capabilities of the nodes. Even if the load balancing mechanism initially assigns the jobs uniformly across available system

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

To address these problems, we have designed dynamic load balancing mechanisms that *redistribute* the jobs across run nodes as needed, to improve overall

system throughput. This redistribution of jobs is different from what is needed by the CAN failure recovery mechanisms. Reallocation of jobs because of nodes failing or departing must always be performed to maintain the CAN properly, while dynamic load balancing is an optional process and is only be used to improve overall system throughput. However, job redistribution (migration) has both benefits and costs. Job migration cost may be higher in a P2P system that spans a wide-area network compared to a local area network, since the job profile has to be transferred, including all input data. For jobs that do not run for a long time, the migration cost may be very high compared to the job execution time. Jobs that run for hours or days can greatly benefit from migration, rather than sitting in a queue for a long time. Therefore, long running jobs having minimal data communication cost are most appropriate for job migration, and fortunately are also the usual characteristics of applications targeted at desktop grid systems. Most long running desktop grid applications, such as those performed by SETI@Home [5] or Folding@Home [34], are indeed long running [101] and are the main target applications for such systems.

We choose to only employ the job migration techniques to jobs that are not currently running, but are currently *waiting* in a queue on a run node, since migrating running jobs requires complex mechanisms for state storage and resuming

the job. Also, in our system, any input data files for a job are transferred to the run node only when the job actually starts running. This means that by targeting only jobs waiting in the queue, the job migration cost is low since we only need to migrate the job description file, which is quite small. We now present our dynamic load balancing schemes, based on either *pulling* jobs to lightly loaded nodes or *pushing* jobs away from heavily loaded nodes.

4.2.1 Models for Migrating Jobs

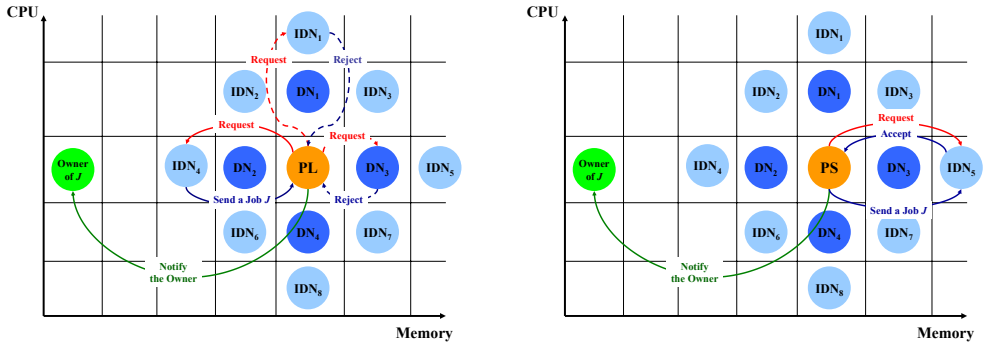
In the push model, a node that has a disproportionate number of jobs in its queue can push jobs to its neighboring nodes in the CAN, while in the pull model a node that becomes idle can pull jobs from its more heavily loaded neighbor nodes. However, the semantics of the matchmaking process and the CAN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

Decentralized dynamic load balancing can be done using the neighbor state information that must be maintained for connectivity in the CAN space. From the perspective of a node, its neighbor nodes are good candidates for running jobs in

its queue, since they are likely to meet the constraints of those jobs, due to the assignment of resource types to the different CAN dimensions. Periodically, and independently of when other nodes send updates, a node sends its own current information (such as zone, coordinates, etc.) and the same information that it currently has for its neighbors in the CAN space to its neighbors [76]. Therefore, each node maintains both the state of its direct neighbors and also state for neighbors of neighbors (indirect neighbors). This information is required to enable the basic CAN failure recovery mechanism, where the node that ends up taking over the zone vacated by a failed neighbor can discover the neighbors of the lost zone through its indirect neighbor information [76]. In our desktop grid CAN, additional load information (i.e., the current size of the job queue) is piggybacked onto the periodic neighbor updates so that each node can estimate the current load of its direct and indirect neighbors.

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

job run times.



(a) Pull Model (b) Push Model

Figure 4.9: Models for Dynamic Load balancing: DN_i and IDN_i denote direct and indirect neighbors, respectively.

PULL Model Figure 4.9 shows two different approaches for the dynamic load balancing of jobs, a pull model (Figure 4.9(a)) and a push model (Figure 4.9(b)). In the pull model, whenever a node PL becomes idle (or very lightly loaded), it tries to pull jobs from its more heavily loaded neighbors (both direct and indirect). In Figure 4.9(a), node PL is performing dynamic load balancing and has twelve neighbors that can be considered. PL sorts those neighbors according to their job queue sizes (propagated through the neighbor updates), and selects the one that has the longest job queue size (but it must be longer than PL's job queue). PL then

sends a message to that neighbor to request a job (*Request* in the Figure 4.9(a)).

One important constraint is that PL must be able to run the job migrated from its neighbor (i.e., it should meet the resource requirements of the job). For that reason, one approach for the pull model is that PL contacts only neighbors that can be covered by its own coordinates (i.e., each coordinate of a neighbor is less than or equal to the corresponding coordinate of PL). For example, in Figure 4.9(a), nodes DN_2, DN_4, IDN_4, IDN_6 and IDN_8 are covered by the coordinates of PL, which means that all jobs in these neighbors are *guaranteed* to be able to run on PL due to the semantics of the multi-dimensional CAN space. While this scheme guarantees meeting the job constraints, it restricts the flow of job migration to only one direction (always from regions closer to the CAN origin than PL). However, the static load balancing mechanism, which pushes jobs away from the node that has the minimum capability to run the job (i.e. meet its resource requirements), may push a job to nodes with higher resource capabilities (Section 4.1) . That means that neighbors that cannot be covered by PL (the other 7 nodes in Figure 4.9(a)) may indeed have jobs that PL can run.

Therefore, in our pull model node PL contacts one of its neighbors based only on their job queue sizes, ignoring their CAN coordinates. However, when the node PL contacts its most heavily loaded neighbor, that neighbor may not have any jobs

waiting in its queue that can be run on PL. In this case, that neighbor simply sends a reject message to PL, and PL tries to pull a job from the next most heavily loaded neighbor node (*Reject* in Figure 4.9(a)). Although this may require several attempts to contact neighbors of PL, the overhead is not too high since the number of neighbors is limited and PL always contact more heavily loaded neighbors than itself (we investigate the overhead incurred by the pull model in Section 4.2.2).

Finally, if PL finds an appropriate node for migration, a job that can be run on PL (selected by searching from the head of the job queue on the node sending the job) is transferred to PL and inserted into PL's job queue (*Send* in Figure 4.9(a)). To ensure fairness among jobs in the system, we sort all jobs in a job queue based on their *submission times* so that migrated jobs may be inserted anywhere in a job queue, not just the end.

PUSH Model In the push model whenever a node PS becomes heavily loaded (i.e. its job queue gets long enough), the node attempts to push one or more of its queued jobs to its more lightly loaded neighbors (both direct and indirect), as seen in Figure 4.9(b). PS sorts those neighbors according to their job queue sizes and picks the one that has the shortest queue size (and the job queue size is shorter than PS's). PS then contacts that neighbor to request job migration. However,

unlike the pull model, node PS never has to make multiple attempts to find a neighbor that can run one of the waiting jobs in its queue. That is because all of nodes that are candidates for job migration from node PS are neighbors of PS, so PS can determine whether a neighbor can run the migrating job before making the request based on the job coordinates (resource requirements) all being less than or equal to the corresponding neighbor coordinates (resource capabilities). Therefore, when PS sorts its neighbors it can safely exclude nodes that cannot run any of the jobs in its queue. This keeps the number of messages the push model requires for performing dynamic load balancing mechanism low, as will be seen experimentally in Section 4.2.2.

Diffusion of Load In both the pull and push models, one important issue is to determine the *idleness* of a node, since jobs should always migrate from heavily loaded nodes to idle ones. If a node N is free (no running or waiting jobs in its queue), we can definitely regard N as an idle node. Therefore, in this case, node N should always try to have a job migrated from one of its neighbors (through either the pull or push mechanism). However, what should happen if N has only one or two jobs in its queue? Note that we calculate job queue size as the number of running *and* waiting jobs in the queue (e.g., a free node has a job queue size of

zero). We could use a threshold to determine the idleness of node *N* so that if *N* has fewer jobs in its queue than the threshold it is regarded as an idle node. However, selecting a good threshold value that is independent of the job characteristics can be very difficult. Therefore, another possibility is for a node *N* to be regarded as idle if and only if it is free (zero queue length). So only free nodes will get jobs migrated from its more heavily loaded neighbors. However, this scheme also may not work well in the decentralized P2P grid environment, because a node shares migratable jobs only with its neighboring nodes. However, if only free nodes are allowed to migrate jobs, that will only balance load in the regions in the CAN space near free nodes, so that jobs may not be propagated over longer distances in the CAN. Therefore, even though all operations are performed locally, a better method would gradually propagate the effects of job migrations so that loads are *diffused* into the entire CAN space.

To achieve that behavior, we employ a *probabilistic* approach for each node to determine whether or not it will accept a migrated job from its neighbors. A node *N* accepts a job migration request with a probability of $\frac{1}{(1+N's\ job\ queue\ size)}$. Therefore, if a node *N* is free, it will always accept migrated jobs from its neighbors. Also, even if *N* has some jobs in its queue, it may get additional jobs migrated from more heavily loaded neighbors. This simple but effective scheme allows jobs to

gradually move from heavily loaded regions to lightly loaded regions in the CAN space, resulting in global diffusion of loads across all available nodes. Note that for all job migrations, the new node must always meet the resource requirements of a migrated job.

Choosing the Best Node for Migration It is possible for a node *N* to receive multiple job migration requests from multiple neighbors at about the same time. For the pull model, that is most likely to occur at the locally most heavily loaded node, and for the push model the most lightly loaded neighbor is likely to have this problem. The solution is for *N* to decide which requesting node finally will get the job. The choice could be done randomly or in order of the requests, but we have designed a method tailored to the characteristics of the load balancing algorithms. For the pull model, if a node receives multiple requests for job migration, it selects the lightest loaded neighbor and sends a job to that node. For this purpose, whenever a node requests a job migration, it includes its current job queue size in the message. Similarly, for the push model if a node receives multiple requests for job migration, it selects its most heavily loaded neighbor. The final step of job migration is to notify the owner node for the migrated job about the migration, so that it can keep track of the run node for the job (as shown in Figure 4.9).

4.2.2 Performance Evaluation

In this section, we evaluate our decentralized dynamic load balancing mechanisms through experimental results obtained via simulations.

Experimental Setup

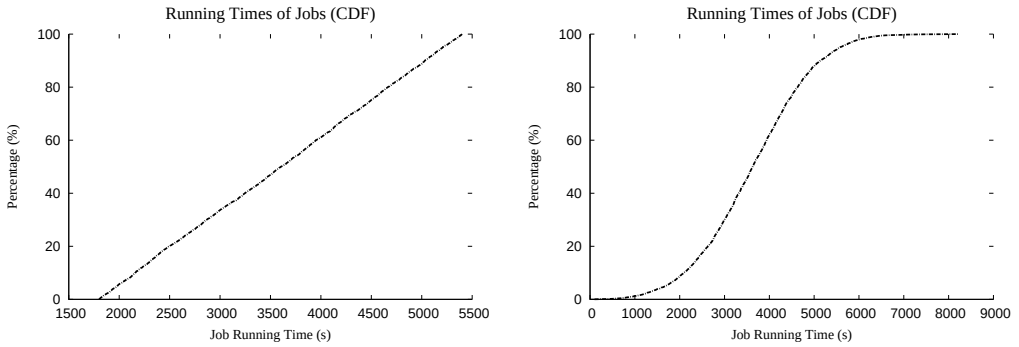
We used five different resource types for nodes and jobs: CPU architecture, operating system type, CPU speed, memory size, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used four different combinations (sub-CANs). Nodes (total 1000 nodes) and jobs (total 5000 jobs) have one of those combinations for their resource specifications and constraints, respectively. We generate continuous resource type values (CPU, memory and disk) for nodes and jobs based on a clustering model, as described in our earlier chapter. We used ten different sets of homogeneous clusters having different continuous resource capabilities, and the resource requirements for jobs are also clustered (i.e., multiple jobs have similar or even identical requirements).

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

(relative to some baseline node CPU speed). Finally, for network communication cost, we model the latency of a packet between any two nodes by an exponential distribution with a mean of 50 milliseconds.

In these experiments, we varied two values, τ for jobs (i.e., average job inter-arrival time) and the *distribution* of job running times. We used two different τ values: 1 and 4 seconds respectively (denoted as *heavy* and *light* workloads). With τ set to 4 seconds, the overall system stays in a steady state, where the rate for incoming jobs and finishing jobs is approximately the same. However, if τ decreases to 1 second, the system becomes heavily loaded and will eventually saturate all available nodes, resulting in indefinite growth of the node job queues. That scenario shows the behavior of our algorithms for dynamic load balancing in a very heavily loaded environment, where the static matchmaking decisions may be made based on stale load information. Also, we used two different distributions for job running times, uniform and normal (as we can see from Figure 4.10). In the uniform model, a job running time is generated uniformly at random from between 30 and 90 minutes and an average of 60 minutes (as seen from Figure 4.10(a)). We also tested the algorithms with normally distributed job running times, with a mean of 60 minutes and a standard deviation of 20 minutes (Figure 4.10(b)). This scenario shows how the algorithms are affected by non-uniformity in job running

times, and also shows the effects of situations where the number of jobs in a node's queue is not a good estimate for the queuing time of a newly assigned job.



(a) Uniform Distribution

(b) Normal Distribution

Figure 4.10: Distributions of Job Running Times

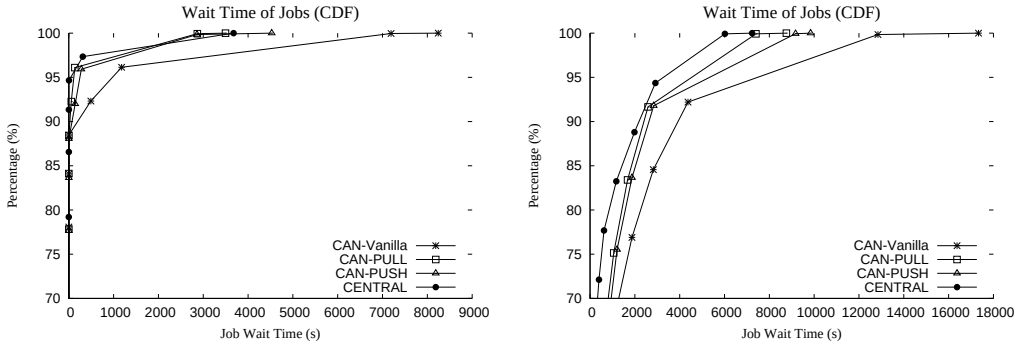
Our metrics are *wait time*, which is the amount of time between when a job is injected by a client and when it actually starts running, and the *rate of dynamic load balancing messaging*, which is the number of messages required to perform the dynamic load balancing scheme per minute. Wait time includes the time to perform the matchmaking algorithm and the time spent waiting in the job queue of a run node before a job is executed. Wait time reflects both protocol overhead and the quality of the matchmaking results, i.e., load balancing. Since the matchmaking cost in our system is very small compared to the job running time [52, 55, 57], the majority of wait time is composed of the queuing time. The

number of dynamic load balancing messages shows the overhead for executing the job redistribution algorithms in a decentralized fashion.

We compare the basic CAN approach that only uses the static job load balancing scheme (labeled as **CAN-Vanilla** in the figures) with the improved CAN approach that uses dynamic load balancing either with the job pull model (**CAN-PULL**) or the job push model (**CAN-PUSH**). Both CAN-PULL and CAN-PUSH perform the dynamic load balancing algorithms every five minutes, which is much longer than the 30 second interval between neighbor updates for CAN maintenance. To see how well the dynamic load balancing schemes work, we also show results for a centralized scheme (**CENTRAL**) that has complete information about the job queue status of all nodes. Similar to our dynamic load balancing mechanisms, CENTRAL periodically redistributes jobs across all nodes in the system. We verified that without the redistribution of jobs, naive Centralized algorithm cannot outperform our CAN-PULL or CAN-PUSH mechanisms.

Experimental Results

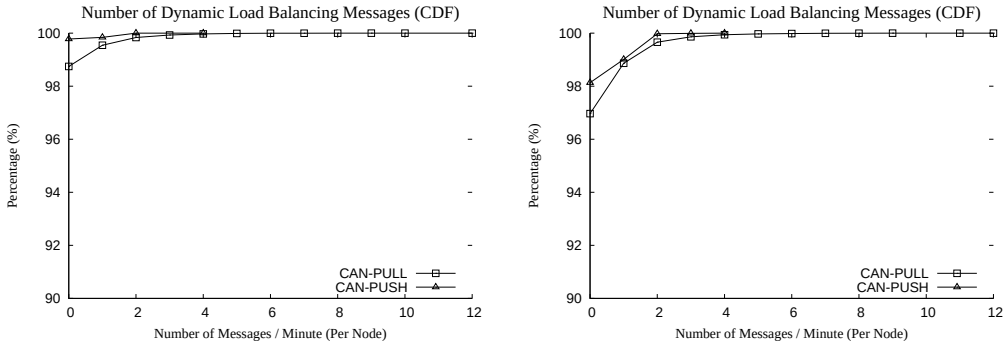
Figures 4.11, 4.12, 4.13, 4.14 and Table 4.1 show performance results for the dynamic load balancing mechanisms, with two different job running time distributions and the two different job inter-arrival times.



(a) Job Wait Time (Light Workload) (b) Job Wait Time (Heavy Workload)

Figure 4.11: Performance Results with Uniformly Distributed Job Running Times: In the figures the Y-axis does not start from 0%, to show the results more clearly

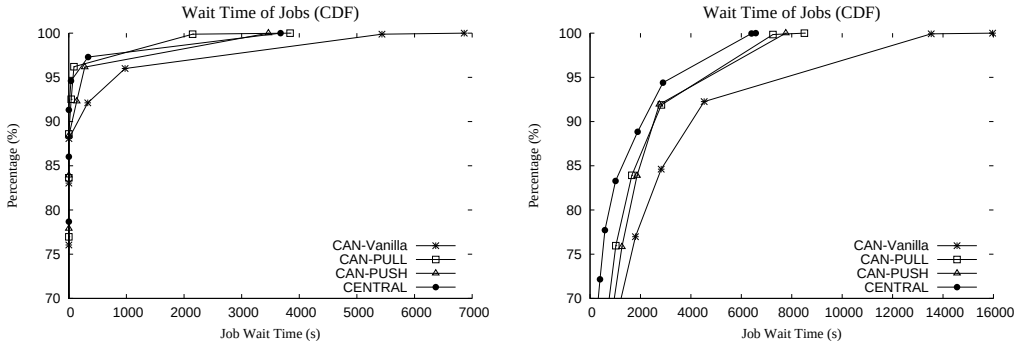
We first discuss the results for uniform job running time distributions, seen in Figures 4.11(a) and 4.11(b). The figures show that the dynamic load balancing schemes (CAN-PULL and CAN-PUSH) greatly improve load balance compared to CAN-Vanilla, and show very competitive performance even compared with CENTRAL. Both CAN-PULL and CAN-PUSH not only remove the high end of the wait time distribution compared to CAN-Vanilla, meaning that the longest waiting jobs wait much less, but also shift the CDF up and to the left, which means that they achieve better distribution of jobs across available nodes compared to CAN-Vanilla. However, under the heavy workloads shown in Fig-



(a) Number of Messages (Light Workload) (b) Number of Messages (Heavy Workload)

Figure 4.12: Overheads with Uniformly Distributed Job Running Times

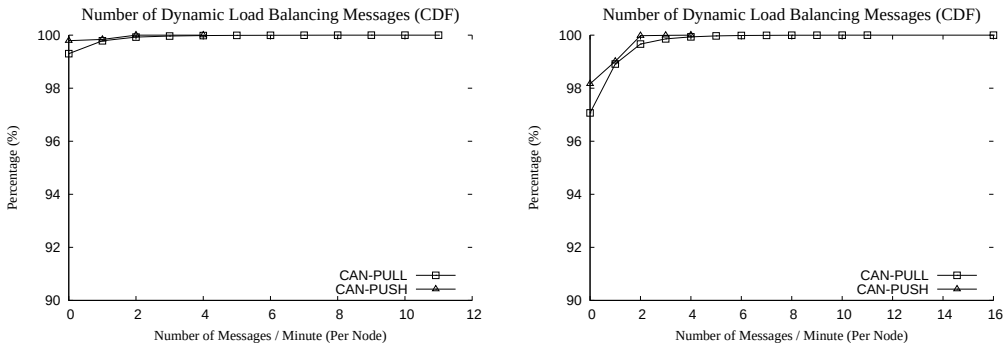
ure 4.11(b), all of the matchmaking frameworks show longer job wait times compared to the lighter workloads. More specifically, the average job wait times for the light workload in Table 4.1(a) show that CAN-PULL and CAN-PUSH decrease the average job wait time to 23% and 36% that of CAN-Vanilla, respectively, while CENTRAL decreases that metric to 22% that of CAN-Vanilla. For the heavy workload, CAN-PULL, CAN-PUSH and CENTRAL decrease the average wait times to 60%, 68% and 44%, respectively, that of the average job wait time for CAN-Vanilla. Therefore, dynamic load balancing algorithms improve load balance for executing jobs dramatically compared to CAN-Vanilla, and show performance close to that of CENTRAL, which has a global view of the entire set of nodes.



(a) Job Wait Time (Light Workload) (b) Job Wait Time (Heavy Workload)

Figure 4.13: Performance Results with Normally Distributed Job Running Times: In the figures the Y-axis does not start from 0% to show the results more clearly

All the benefits from dynamic load balancing come with additional cost (overhead), since redistribution of jobs occurs periodically. However, as can be see from Figure 4.12(a) and 4.12(b), the networking requirements to perform dynamic load balancing are very low, totaling only a few messages per minute. The messages counted include all those needed to perform the dynamic load balancing algorithms, which include contacting neighbor nodes to request a job migration, actually migrating a job, and notifying the job owner node of the new run node (as described in Section 4.2.1). One important characteristic about Figure 4.12(a) and 4.12(b) is that the graphs show the number of messages over the *entire* simulation, which means that for *all* one minute intervals simulated, there was no



(a) Number of Messages (Light Workload) (b) Number of Messages (Heavy Workload)

Figure 4.14: Overheads with Normally Distributed Job Running Times

node in the system that processed more than 12 dynamic load balancing messages (for CAN-PULL). When we measure the total size of the dynamic load balancing messages across all the workloads, CAN-PUSH and CAN-PULL send up to 300 bytes and 600 bytes per minute respectively. As was described in Section 4.2.1, CAN-PULL generates more messages than CAN-PUSH, since a node can perform multiple retrials to contact its neighbors to find a job that can be run on that node. In results not shown, we also measured the average number of messages sent in the CAN per minute during the entire simulation and CAN-PULL on average causes only 0.3% of all messages (meaning the vast majority come from other sources, including CAN maintenance and matchmaking), while CAN-PUSH causes only 0.2% of the total number of CAN messages.

	Vanilla	*PULL*	*PUSH*	*CENTRAL*
Light	138.8	31.8	49.8	30.8
Heavy	1236.5	740.2	843.8	547.3

(a) With Uniformly Distributed Job Run Times

	Vanilla	*PULL*	*PUSH*	*CENTRAL*
Light	123.5	19.9	41.5	34.2
Heavy	1261.0	758.9	806.2	526.6

(b) With Normally Distributed Job Run Times

Table 4.1: Average Job Wait Time (seconds)

The results presented in Figure 4.13 and Table 4.1(b) show that our dynamic load balancing mechanisms are effective regardless of the distribution of job running times. For the light workload (Figure 4.13(a)), CAN-PULL and CAN-PUSH decrease the average job wait time to 16% and 34% that of CAN-Vanilla, respectively, while CENTRAL decreases that metric to 27.7% that of CAN-Vanilla. Occasionally, CENTRAL shows somewhat longer job wait times than CAN-PULL because it assigns jobs to the fastest idle node in the system, so some jobs requiring high resource capabilities that arrive after one or more jobs have already been

assigned to the highly capable node may wait in its queue. For the heavy workload (Figure 4.13(b)), CAN-PULL, CAN-PUSH and CENTRAL decrease the average wait time to 60%, 64% and 42% that of CAN-Vanilla. Of greater interest is that CAN-PULL and CAN-PUSH perform better for this metric for the normal job running time distribution compared to the uniform distribution. That is because job running times have much wider variance for the normal distribution, so that the number of jobs in the queue is definitely not a good metric for the load balancing scheme. Again, the cost to perform dynamic load balancing is still low, as can be seen in Figure 4.14(a) and 4.14(b). If we measure the average number of dynamic load balancing messages per minute across the entire simulation, CAN-PULL again causes only 0.3% of the average number of all CAN messages, while CAN-PUSH causes 0.2% of all messages, the same as for the uniform distribution of job running times. That is because the overhead of the CAN-PULL or CAN-PUSH algorithms is effectively independent of the distribution of job running times.

Another interesting result is that, across all of workload combinations (different loads and job running times), CAN-PULL provides better load balance (measure by job wait times) than CAN-PUSH. We believe that is because in CAN-PULL idle (or comparatively lightly loaded) nodes aggressively pull jobs

from their more heavily loaded neighbors, compared to the idle nodes in CAN-PUSH passively accepting jobs from their neighbors. This is similar to Demers et al. [29] suggestion to use a pull or combined push-pull approach rather push when epidemic algorithms are used for distributing updates and driving replicas toward consistency in distributed database systems. Although our problem does not exactly match their model, we can regard the distribution of jobs across multiple nodes as spreading information throughout the set of distributed nodes. However, since CAN-PUSH has the advantage of lighter overhead (counting messages) compared to CAN-PULL, both dynamic load balancing approaches have their strengths and weaknesses. So if the target system can handle some extra messages, CAN-PULL is the best choice. Otherwise CAN-PUSH should be used in to perform dynamic load balancing, since it can still greatly improve overall system throughput compared to CAN-Vanilla, which does not do any dynamic load balancing.

4.3 Summary

In this chapter, we have presented our comprehensive load balancing techniques that can initially assign jobs to available heterogeneous computational resources

based on pushing mechanism and later redistribute them if need, by employing decentralized dynamic load balancing mechanisms.

We improved our static load balancing scheme for executing jobs by pushing jobs into underloaded regions of the CAN space. Nodes periodically send load information towards the origin in each CAN dimension. This information is aggregated at each step, resulting in each node having partial information about load in all regions of the CAN space containing nodes more capable,– exactly those nodes that are also able to run the node's jobs. In times of high load, a node can therefore push jobs towards regions of high capability and low load, based solely on local information. Note that our pushing mechanism and load aggregation process are only applied along the continuous resource dimensions. As we discussed in Section 3.2, we integrate all types of resource by configuring separate sub-CANs for each combination of categorical resource types and inside each sub-CAN, we do the load balancing along the continuous dimensions. Therefore, overall matchmaking and load balancing procedure can be illustrated as shown in Figure 4.15 (revising the Figure 3.6).

However, we found potential load imbalance problems that arise from our static load balancing mechanisms for assigning jobs to nodes that can arise for various reasons, including the heterogeneity of the available nodes or the jobs to be

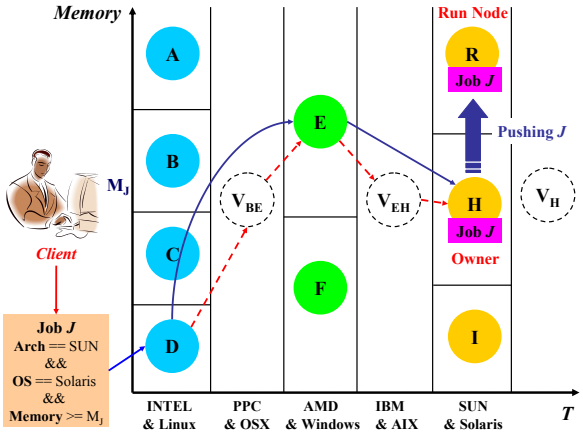


Figure 4.15: Overall Matchmaking and Load Balancing Process: Solid arrows denote the physical routing path of job *J*, while dotted arrows show the logical routing path.

run, and from stale information in the P2P system. Therefore, we greatly improve our load balancing scheme by providing lightweight yet effective dynamic load balancing mechanisms to overcome load imbalances caused by the limitations of the initial static job assignment scheme. Throughout extensive simulation results, we showed that dynamic load balancing makes the overall system more scalable, by improving overall system throughput and response time with low additional overhead.

Chapter 5

Reducing the System Load

In this chapter, we describe a set of optimizations to reduce the overheads in the system and distribute them fairly by employing modified heartbeat messaging and randomizing job ownership. First, we address the increased amount of heartbeat message exchanges between virtual peer and neighbors discussed in Section 3.2.3. Then, we reduce the cost of monitoring job executions by effectively randomizing ownership of jobs.

5.1 Modified Heartbeat Messaging

As we discussed in Section 3.2.3, our virtual peer-based resource integration scheme has a scalability issue since a virtual peer maintains a large number of neighbors in sub-CANs. Each time when a virtual peer sends a heartbeat message

to its neighbors, it packs *all* of its neighbor information in the update message and sends it to *all* of its neighbors. This is because each node in CAN sends its own information and its neighbor information in a periodic update message [76]. This does not only increase the *size* of a single update message but also increase the *number* of messages sent by the virtual peer. Therefore, such messages can add substantial overhead for the nodes responsible for the virtual peers. As an example, if a virtual peer manages 1000 neighbors then the size of a single heartbeat message is about 600KB and this can become a significant burden since a virtual peer sends update messages to its all neighbors (i.e., the total size of the messages would be 600MB at every update).

To deal with the large size of a periodic update message, we introduce a *partial update* mechanism for heartbeat messaging. Whenever a node sends information about its neighbors, it may only send partial neighbor information. For this purpose, we use a threshold value, *PU_Threshold*, which limits the number of neighbors that are included in a periodic update message in each *direction* (upper or lower in each dimension). Therefore, even with only partial information about neighbors, each node will let its neighbors know about at least one and at most PU_Threshold neighbors in each direction (we select the PU_Threshold neighbors from each direction randomly). This ensures the correctness of the failure recov-

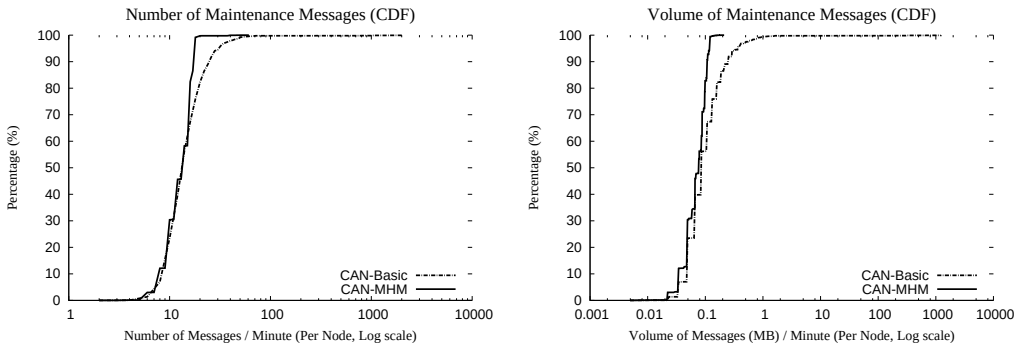
ery algorithms, so that whenever a node leaves the system or fails, the neighboring nodes can determine the neighbors of the lost zone through the neighbor information maintained in that direction. Note that the partial updates mechanism does not affect failure recovery along the T dimension, since a physical peer never takes over the zone along that dimension (only virtual peers do as described in Section 3.2). By employing partial update mechanism, the size of a single update message is limited to $d * 2$ (for both directions) * $PU\_Threshold * SN$ where d is the number of dimensions and SN denotes the size of a single neighbor information. A single neighbor information includes zone, coordinates and neighbors of neighbors (indirect neighbors) information, etc. Therefore, we can reduce the system load and distribute it more fairly among available system nodes.

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

update mechanism described previously, which only lengthens the average time between heartbeat messages sent to each neighbor. By employing this round-robin heartbeat messaging scheme, the number of messages sent per period is limited to the 2 * d so that every node in our CAN sends the same number of messages as the original CAN [76].

5.1.1 Effects of Modified Heartbeat Messaging

To see the effects of our modified heartbeat messaging scheme in terms of system maintenance cost, we use similar experimental setup (1000 nodes and 5000 jobs in the system) as described in earlier chapters using an event-driven simulator. We use the same set of resource types, CPU architecture, operating system type, CPU speed, memory, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used two different combinations to show the behavior of our system with a large number of jobs and fewer available sub-CANs resulting in the number of neighbors of a virtual peer tends to increase. Again, we used ten different sets of homogeneous clusters having different amounts of continuous resource capabilities and resource requirements of jobs are also clustered. Finally, each node (including both of physical and virtual peers) refreshes its neighbors with heartbeat messages every 30 seconds.



(a) Number of Messages

(b) Volume (MB) of Messages

Figure 5.1: Effects of Modified Heartbeat Messaging: Note that X-axis is in *log scale*

In this experiment set, our metrics are *number* of messages and *volume* of messages (i.e., size of messages) in the system. Specifically, to see the effects of our modified heartbeat messaging scheme, we investigate the *maintenance* message distribution which includes the messages required to maintain the CAN DHT. We test the CAN approach using normal periodic heartbeat messaging scheme (**CAN-Basic**) and the improved CAN approach employing our modified heartbeat messaging techniques (**CAN-MHM**).

Figure 5.1 shows the effects of modified heartbeat messaging in terms of system maintenance cost. Our modified heartbeat messaging scheme includes two optimizing mechanisms: round-robin heartbeat messaging to reduce the number

of messages and partial updates to reduce the size of each message. By employing these techniques, CAN-MHM effectively reduces both the number of messages per minute (Figure 5.1(a)) and the volume of messages per minute (Figure 5.1(b)). CAN-MHM not only removes the high end of the number (volume) of maintenance messages compared to CAN-Basic, meaning that there is no node in the system that processed much larger amount of maintenance work, but also shift the CDF up and to the left, which means that they achieve better distribution of system loads across available nodes compared to CAN-Basic. This is because in CAN-Basic, virtual peers maintaining a large number of neighbors in sub-CANs incur huge overheads during exchanging periodic update messages with neighbors. This means that some nodes managing these virtual peers will be very heavily loaded only because of processing periodic update messages. This overloaded system maintenance cost is not sustainable in the P2P desktop grid system since every node in our system is a peer so that unfair distribution of system loads cannot attract the participation of desktop machines into our system.

Total messages in our system not only include CAN maintenance messages but also other job-related messages such as matchmaking messages and node join messages. However, in terms of average number (volume) of messages per minute, overall 99% of total messages come from CAN maintenance as we can

	Total	*Maintenance*	*Others*
CAN-Basic	19.4	19.2	0.2
CAN-MHM	13.6	13.4	0.2

(a) Average Number of Messages

	Total	*Maintenance*	*Others*
CAN-Basic	1575.6	1575.5	0.1
CAN-MHM	74.7	74.4	0.3

(b) Average Volume (KB) of Messages

Table 5.1: Average Number/Volume of Messages (Per Minute, Per Node)

see from Table 5.1. Therefore, by employing our modified heartbeat messaging scheme, our system can get rid of most overheads coming from CAN maintenance and becomes more scalable and effective.

5.1.2 Performance Implications of Modified Heartbeat Messaging

Although employing modified heartbeat messaging scheme can greatly reduce the overall system maintenance cost and distribute the load fairly across nodes in

the system, it can affect the quality of load balancing since we are using partial neighbor information. Therefore, in this section we investigate the performance effects and implications of the modified heartbeat messaging scheme.

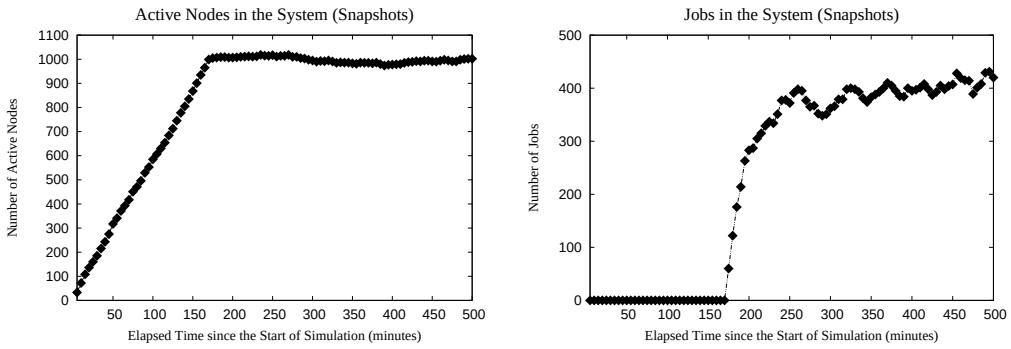
Again, we use a similar experimental setup using our event-driven simulator where there are five different resource types, CPU architecture, operating system type, CPU speed, memory, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used two different combinations. So nodes and jobs can belong to either of the combinations with respect to their resource specifications and constraints, respectively. We used ten different sets of homogeneous clusters having different amounts of continuous resource capabilities and resource requirements of jobs are also clustered. As described in previous chapters, our algorithms can also handle truly heterogeneous set of nodes and jobs where there are few identical nodes (an unclustered set of nodes). However, we use the clustered model for our experiments since this is a likely scenario and it also shows the behavior of the system when nodes have many neighbors along the virtual dimension, as described in Section 3.2.3.

One important characteristic of the test workload is that the overall system reaches a steady state, for both available nodes and active jobs, during the simulation period (as seen from Figure 5.2). The way we generated the workload is

that first an initial set of 1000 nodes join the system. Then, new node join events and existing node departure events (graceful leaving or failure) occur at approximately the same rate. Once the system reaches the steady state in terms of active nodes a total of 5000 jobs are submitted. Again, the system behavior is measured in a steady state, so that the number of active jobs remains about the same (jobs arrive and complete at about the same rate), and we show the performance of each matchmaking mechanism in this steady state to avoid the transient effects of earlier jobs that see a largely empty system.

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

We test the CAN approach both before addressing scalability issues (**CAN-Basic**) and the improved CAN approach employing partial updates mechanism. For partial updates, we used two different PU_Thresholds, 1 and 2 (**CAN-PU1** and **CAN-PU2**, respectively). Since both thresholds showed similar (good) behavior,



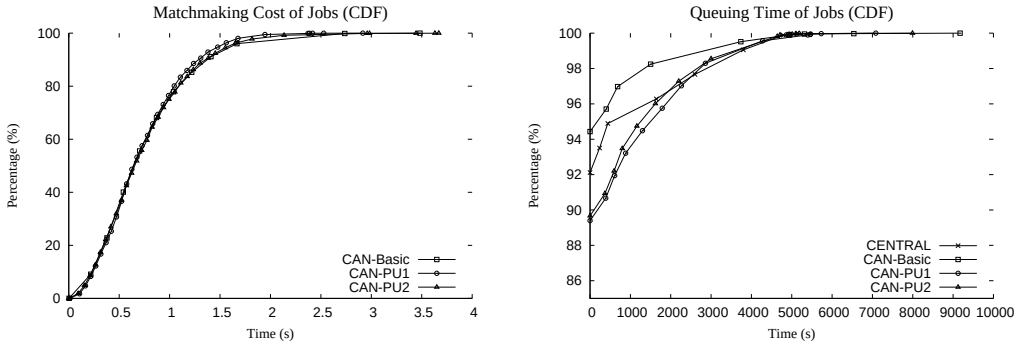
(a) Steady-state behavior of grid nodes

(b) Steady-state behavior of queued and running jobs

Figure 5.2: Populations of Available Nodes and Jobs over Time

we do not show results for higher threshold values. To see how well the workload could be balanced, we also show results for a centralized scheme (**CENTRAL**) that uses knowledge of the status of all nodes and jobs.

Figure 5.3 shows the matchmaking cost and queuing time of jobs with the different algorithms. As we can see from Figure 5.3(a), the CAN-based mechanisms can match the jobs having different resource constraints with available heterogeneous resources with very low cost. Most of the jobs can be matched within a couple of seconds, but some take much longer. High matchmaking costs can occur when new nodes join or existing nodes leave the system Those events cause transient system states where matchmaking for a job cannot proceed un-



(a) Matchmaking is not noticeably impacted by partial updates

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

Figure 5.3: Matchmaking and Queuing Time

til the holes in the CAN space caused by the node departures have been repaired. During those periods CAN routing can fail, so has to be retried. However, jobs are still matched within a very short time period, since the system quickly recovers from those transient states.

Figure 5.3(b) shows the quality of load balancing of the CAN-based approaches compared to the centralized matchmaker. As the figure shows, all of the CAN-based frameworks show performance competitive with CENTRAL, although there are some jobs that wait longer in the queues. Through the experiments shown here, we verify that the partial update mechanism does not affect the quality of load bal-

ancing very much, so that our algorithms perform well for both categorical and continuous resource capabilities for nodes and requirements for jobs.

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

5.2 Randomizing Job Ownership

As noted in Section 2.1, for each job our system has two nodes that know about it, an owner and a run node, to enable reliable recovery from various types of failures. In prior work, we have mainly addressed load balancing of jobs across run nodes, which directly affects overall system throughput and response time. However, we have discovered that balancing load across *owner* nodes is also important to evenly balance overall system maintenance costs across all nodes.

A node o becomes the owner of a job j if and only if its zone contains the point for j (determined by j's resource requirements). There are two main reasons for assigning an owner in this way. First, it enables efficient matchmaking for j to find a run node, since neighbors in the CAN of the owner zone become good candidates for running the job (since many of them will meet the resource requirements of the job because of the CAN organization). Therefore, our pushing mechanism (Section 4.1) basically starts from the region consisting of minimally capable nodes for running the job, and searches for more capable nodes for better load balancing. Second, we require a deterministic mechanism to decide the job owner, since an owner node can depart the system at any time (or fail). If that occurs, the run node or a client always can find the new owner node by routing again to the job's coordinates, since the basic CAN recovery mechanisms will assign the

zone to another node.

However, this mechanism can cause high overhead for a small set of nodes that own a disproportionate fraction of the jobs, because of the heartbeat messages that run nodes periodically exchange with owner nodes (so that either can detect failure of the other). As an extreme example, suppose that the zone for node n contains the origin in the CAN space. If clients submit 1000 jobs that specify no resource requirements at all, then n will become the owner of all the jobs (since the default value in each resource dimension is 0). If all of the jobs run for a long time, node n will receive thousands of messages periodically from the multiple run nodes that have been assigned to run the jobs. Also, if n departs the system, all the jobs that n owned must be redistributed to the neighbors of n in the CAN space, resulting in high recovery cost. This message load balancing problem can occur whenever many jobs have very similar or even identical resource requirements, resulting in hot spots (in terms of owning jobs) in the CAN space. The node(s) in a hot spot will have much higher message loads, and therefore higher overhead costs for participating in the desktop grid, than other nodes.

In the following sections, we address these problems and propose an effective mechanism to uniformly distribute ownership of jobs so that the number of owned jobs per node does not depend on the distributions of resource capabilities of

nodes or resource requirements of jobs in the system.

5.2.1 Random Walking along *T* dimension

One straightforward approach to address the problem of job ownership is randomizing each coordinate of a job (from 0 to some maximum value *M*) and use them to determine the owner node (by routing to the randomly generated coordinates). Thus, each job has two different sets of coordinates: randomly generated *pseudo* coordinates and *real* coordinates (corresponding to the job's resource requirements). Pseudo coordinates are used only for determining owners and real coordinates are used for matchmaking process. For this purpose, we may propagate the maximum value of each resource dimension (*M*) through the periodic load aggregation mechanism (presented in Section 4.1) and use it to set each range of resource dimensions. However, we cannot assume that resource capabilities of nodes are uniformly distributed in the CAN space (as we discussed in Section 4.1 and Figure 4.2). This means that randomly generated coordinates still can result in a biased job ownership distribution unless we know the exact distribution of resource capabilities of all nodes in the system. This is almost impossible to get in a decentralized desktop grid system where no centralized information exists.

Figure 5.4 shows basic concepts of our approach where there are four different

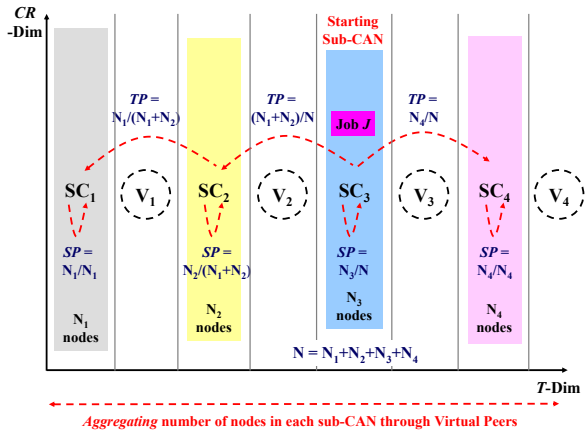


Figure 5.4: Randomizing Ownership of Jobs: *CR* denotes the continuous resource dimensions.

sub-CANs (SC_i's) in the system (i.e., four different combinations of categorical resource types such as architecture and operating system type). We first distribute the ownership of jobs fairly across multiple sub-CANs (i.e., make the probability for each sub-CAN to own a job be proportional to the number of nodes in that sub-CAN) and then determine the ownership of a job inside the sub-CAN randomly. Let P_i be the probability for SC_i to have an owned job and N_i be the number of jobs in the SC_i. Then, the total number of nodes in the system (denoted by N) is the sum of all N_i's (in the figure, $N=N_1+N_2+N_3+N_4$). To uniformly distribute the job ownership across multiple sub-CANs, P_i should be $\frac{N_i}{N}$ since each sub-CAN may have different number of nodes. The question is then how we can distribute

jobs across multiple sub-CANs in a decentralized fashion and based only on local information.

To address this problem, we employ *random walking* along the T dimension based on *aggregated* information propagated through virtual peers. For the job pushing mechanism, our system periodically aggregates load information along each continuous resource dimension (Section 4.1) which is piggybacked on the conventional CAN periodic neighbor update mechanism [76]. Similarly, each virtual peer sends the number of physical peers in its left and right sub-CANs (along T dimension) and this information is aggregated (bidirectional aggregation of the number of nodes). This enables each physical peer can estimate the number of nodes in its left sub-CANs, right sub-CANs and its own sub-CAN where it belongs to through the information provided by the virtual peers.

Based on this aggregated information about the number of nodes in the system, random walking across multiple sub-CANs can be effectively performed along T dimension. There are three important factors that affect behaviors of random walking: *transition probability* (TP), *stopping probability* (SP) and *direction*. The TP is the probability to move from current sub-CAN to another sub-CAN (left or right) and the SP is the probability to stop walking at current sub-CAN. TP and SP can be calculated based on the aggregated number of nodes in other sub-

CANs, the number of nodes in current sub-CAN and the direction of walking. The direction (left or right walking along *T* dimension) is determined at the initial step of walking and it never changes until the random walking stops at some point (no coming back). For example, in the Figure 5.4, job *J* starts from SC_3 and there are three different choices for this job, start walking into left (right) direction or stops at current sub-CAN. The left (right) transition probability for this job becomes [Number of nodes in left (right) sub-CANs / Total number of nodes in the system]. In the example, this probability becomes either $\frac{N_1+N2}{N}$ (for left walking) or $\frac{N_4}{N}$ (for right walking). Also, with the probability of [Number of nodes in current sub-CAN / Total number of nodes in the system], this job can stop at current sub-CAN ($\frac{N_3}{N}$ in the example).

This simple probabilistic walking model exactly matches with our previous uniform distribution model, where each sub-CAN SC_i has the probability of owning a job as $\frac{N_i}{N}$. Once the walking starts into a specific direction, it never comes back to the other direction and transition/stopping probabilities can be calculated in a similar way. However, at this time, we only have two choices, whether keep moving into another sub-CAN based on the determined direction or just stop the walking (Figure 5.4 shows the TP and SP of each step of walking along *T* dimension).

Until now, we have presented our probabilistic model of random walking along T dimension to select one of the sub-CANs randomly. Now, we have to decide the owner node of a job *inside* the determined sub-CAN and this should be done also uniformly at random. One of the important characteristics of our routing mechanism is that whenever a request is delivered from one sub-CAN to another sub-CAN, it utilizes neighbors of a virtual peer (specialized routing in T dimension described in Section 3.2.3). Each physical peer directly sends a request from a sub-CAN to another sub-CAN through one of the neighbors of a virtual peer (which is *randomly* selected). This prevents all routing requests delivered from one sub-CAN to another sub-CAN from always going through a node managing virtual peers. This special routing mechanism can be effectively combined with our new algorithm for determining random owners. Since the routing request is sent to one of nodes in target sub-CAN randomly (selected from a virtual peer's neighbors), every time a routing request is processed across multiple sub-CANs, it may go through different destination nodes. Therefore, whenever the walking along T dimension stops at a specific sub-CAN, current node simply can become the owner of the job (since it is randomly selected inside this sub-CAN during routing procedures).

The final step is to generate random (pseudo) coordinates for a job, that lie

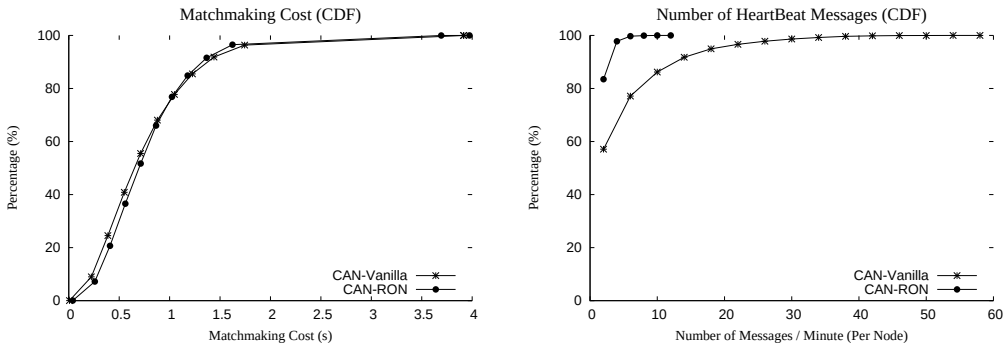
in the zone of the selected owner node, but these coordinates are used only for selecting the owner. After determining a random owner for the job, the match-making process starts from that point, i.e., route to the node whose zone contains the real coordinates of the job (its resource requirements) and perform the pushing mechanism in the CAN to balance load across eligible run nodes.

5.2.2 Effects of Randomizing Ownerships

To see the effects of randomizing job ownership, we use similar experimental setup (1000 nodes and 5000 jobs in the system) as described in earlier sections, using an event-driven simulator. We use the same set of resource types, CPU architecture, operating system type, CPU speed, memory, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used two different combinations to show the behavior of our system with a large number of jobs and fewer available sub-CANs resulting in the number of owned jobs per node tends to increase. Again, we used ten different sets of homogeneous clusters having different amounts of continuous resource capabilities and resource requirements of jobs are also clustered.

In this experiment set, our metrics are *matchmaking cost* (the amount of time between when a job is injected and when it is assigned to a run node) and *rate*

of heartbeat messaging (the number of heartbeat messages exchanged between owner and run nodes per minute), and *number of owned jobs per node* (during entire simulation). Matchmaking cost directly quantifies the overhead needed to perform the matchmaking in a decentralized manner and includes the time to find a random owner for a job (in our new algorithm). Number of heartbeat messages and owned jobs per node can show whether there are any hot spots in the system where some nodes own a disproportionate number of jobs. We test the CAN approach both before addressing job ownership problems (**CAN-Vanilla**) and the improved CAN approach employing our new techniques for randomizing job owners (**CAN-RON**).



(a) Matchmaking Cost (b) Number of Heartbeat Messages

Figure 5.5: Costs and Benefits of Randomized Owners

Figure 5.5 and 5.6 show the behavior of our system with improved mechanism

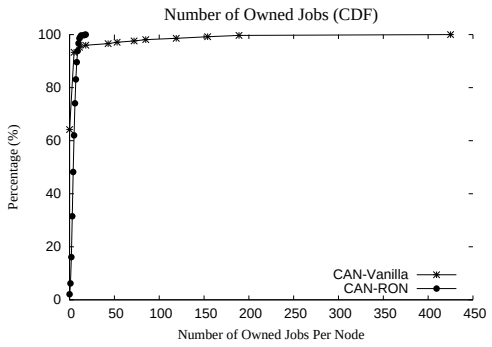


Figure 5.6: Number of Owned Jobs Per Node

for randomizing job ownership. Figure 5.5(a) shows that in terms of matchmaking cost, CAN-RON does not have any significant overhead even though it has two steps of matchmaking process (first finding the random owner of a job and then do the conventional matchmaking and load balancing). This is because additional overhead comes only from traversing multiple sub-CANs (random walking) and the rest of procedures is essentially same since even in CAN-Vanilla, a job can start from anywhere (injection node) in the system. However, if we increase the number of sub-CANs, it can affect the overall matchmaking cost of CAN-RON. In the experiments not shown here, we have verified that with more than two sub-CANs, the cost is still negligible.

With this minimal cost, CAN-RON can achieve much better load balancing of job ownership as we can see from Figure 5.5(b) and 5.6. In terms of number of

heartbeat messages exchanged between owner and run nodes, CAN-RON shows much less overhead compared to CAN-Vanilla (as seen from Figure 5.5(b)). This is because in CAN-Vanilla, a small number of nodes may become owners of many jobs since in our workload, job requirements are clustered. Although the virtual dimension can help to distribute the ownership across multiple identical nodes, it depends on how many such nodes are existing in the system (as we mentioned an extreme case earlier). Therefore, CAN-Vanilla's ability to distribute job ownership is heavily dependent on the distribution of node capabilities and job resource requirements. Figure 5.6 shows the number of owned jobs per node during *entire* simulation. It shows that there are some nodes who own more than 400 jobs in the system and if all of jobs are running very long time (several hours or even for days), these nodes will receive a large number of heartbeat messages from run nodes. Also, if these critical nodes depart the system, hundreds of jobs should be redistributed resulting in high recovery cost compared to CAN-RON.

To summarize, CAN-RON could achieve much better distribution of job ownership which is *not* dependent on any of node capabilities or resource requirements of jobs and reduce the overhead of monitoring jobs effectively.

5.3 Summary

In this chapter, we have discussed our techniques to reduce the overall system maintenance cost and distribute the system load fairly across available nodes in the system.

First, we addressed the problem of periodic neighbor update message exchanges between virtual peers and physical neighbors by employing modified heartbeat messaging scheme. Throughout our improvements, we could not only reduce the size of a single update message (partial updates mechanism) but also reduce the number of messages exchanged (round-robin heartbeat messaging) without affecting the correctness of our basic CAN. As the experiments showed, majority of the messages in our desktop grid system comes from maintaining CAN space so that by reducing the cost for CAN maintenance, we could greatly improve the scalability of our system.

Second, we noticed that the load balancing of job ownership is also important since some nodes can own a disproportionate fraction of the jobs in the system which results in high costs for message exchanges and failure recovery. By effectively randomizing the job ownership in a decentralized manner, we could achieve much better distribution of job ownership which is not dependent on any of node capabilities or resource requirements of jobs.

All of our efforts to minimize any overheads in the system is to ensure the *fairness* among the participating nodes in our P2P desktop grid. A truly scalable P2P desktop grid computing system must be able to balance the load of job executions and introduce minimal and fair overheads among the nodes in the system. Throughout our comprehensive load balancing mechanisms presented in Chapter 4 and a set of optimizations to minimize overheads in the system presented in this Chapter, we could build a scalable and effective P2P desktop grid system.

Chapter 6

Large Scale Experiment

In this chapter, we evaluate our system with a large set of node populations and job workloads to see whether the system scales gracefully as more nodes and jobs are injected. The evaluated system under this experimental set includes everything discussed in this dissertation: comprehensive load balancing techniques based on pushing mechanism and pull-based dynamic load balancing (as described in Chapter 4), and a set of optimizations to minimize the system load based on modified heartbeat messaging and randomized job ownership (as described in Chapter 5).

6.1 Experimental Setup

Similar to the previous experiments, we used five different resource types for nodes and jobs: CPU architecture, operating system type, CPU speed, memory size, and disk space. For the categorical resource types (architecture and operating system), the nodes and jobs used four different combinations (sub-CANs). Nodes (total 10000 nodes) and jobs (total 50000 jobs) have one of those combinations for their resource specifications and constraints, respectively. We generate continuous resource type values (CPU, memory and disk) for nodes and jobs based on a clustering model, as described in Chapter 3.1.2. We used 100 different sets of homogeneous clusters having different continuous resource capabilities, and the resource requirements for jobs are also clustered (i.e., multiple jobs have similar or even identical requirements).

In this experiment, τ for the jobs (average job inter-arrival time) is set to 200 milliseconds to achieve the steady state where the rate for incoming jobs and finishing jobs is approximately the same (as seen from Figure 6.1). Since the number of available nodes increased, we had to reduce the average job inter-arrival time to make the overall system in the steady state (e.g., 4 seconds for τ was used in Section 4.2.2). A job running time is generated uniformly at random from between 30 and 90 minutes with an average of 60 minutes. As in the experiments

in Section 4.2.2, CAN performs the dynamic load balancing mechanism based on pull model every five minutes, which is much longer than the 30 second interval between neighbor updates for CAN maintenance. Finally, for the network communication cost, the average latency of a packet between any two nodes in the system is set as 50 milliseconds with an exponential distribution.

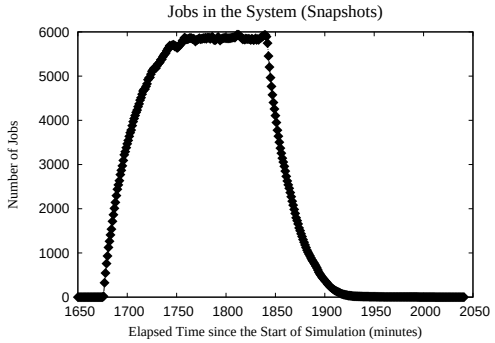
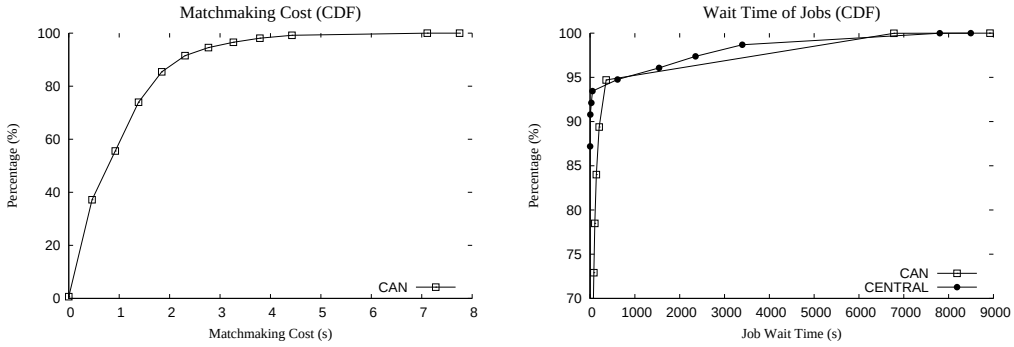


Figure 6.1: Population of Available Jobs in the System

We compare our CAN-based approach (labeled as **CAN** in the figures) with a centralized scheme (**CENTRAL**) that has complete information about the job queue status of all nodes. Similar to the previous experiments discussed in Section 4.2.2, CENTRAL periodically redistributes jobs across all nodes in the system.



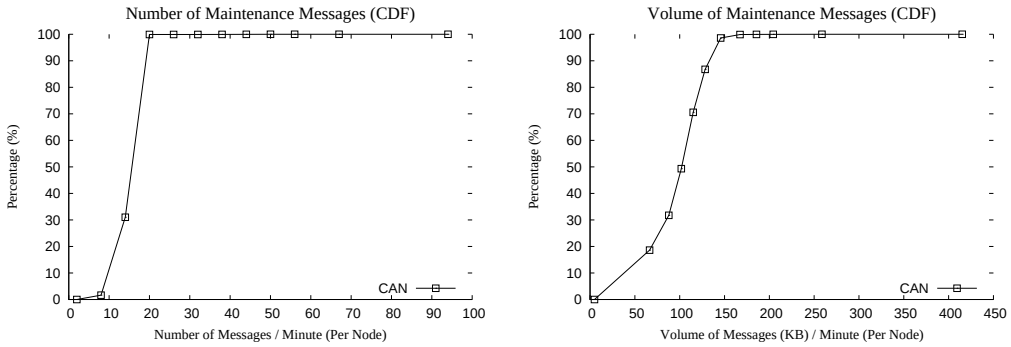
(a) Matchmaking Cost (b) Wait Time of Jobs

Figure 6.2: Performance Results (Matchmaking Overhead and Load Balancing): Note that in the Figure 6.2(b), the Y-axis does not start from 0% to show the results more clearly

6.2 Experimental Results

Figure 6.2 shows the performance results of CAN in terms of decentralized matchmaking overhead and quality of load balancing compared to CENTRAL. As we mentioned earlier, wait time reflects both protocol overhead and the quality of the matchmaking results, i.e., load balancing. Since the matchmaking cost in our system is very small compared to the job running time, the majority of wait time is composed of the queuing time. As we can see from Figure 6.2(b), CAN shows still very competitive performance in terms of load balancing compared to CENTRAL by employing our comprehensive load balancing techniques. The cost for

performing decentralized matchmaking has slightly increased due to the increased number of available nodes in the system compared to the experimental set with 1000 nodes (Figures 5.3(a) and 5.5(a)). However, as we increase the number of nodes and jobs by a factor of 10, the average matchmaking cost increased only 30%. Therefore, the overall system scales gracefully as more nodes and jobs are injected.



(a) Number of Messages

(b) Volume (KB) of Messages

Figure 6.3: Distribution of Maintenance Messages in CAN

As we described in Chapter 5, we have described a set of optimizations to reduce the overheads in the system and distribute them fairly by employing modified heartbeat messaging and randomizing job ownership.

We first discuss the results for performing modified heartbeat messaging with a large set of nodes and jobs in the system. As we can see from Figure 6.3 and

Total	*Maintenance*	*Others*
15.7	15.5	0.2

(a) Average Number of Messages

Total	*Maintenance*	*Others*
99.7	99.5	0.2

(b) Average Volume (KB) of Messages

Table 6.1: Average Number/Volume of Messages (Per Minute, Per Node)

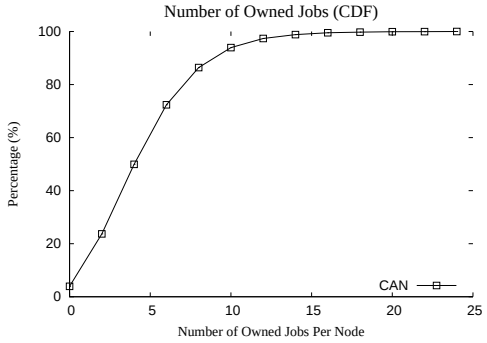


Figure 6.4: Number of Owned Jobs Per Node

Table 6.1, overall maintenance overhead of CAN is not heavily affected by the increased number of available nodes or jobs in the system. Figures 6.3(a) and 6.3(b) show the number (volume) of maintenance messages sent per minute during the *entire* simulation. More specifically, in 99.9% of all one minute intervals simulated, there was no node in the system that processed more than 20 messages or 180KB. There are some intervals where some nodes processed more messages and

larger volumes but these events mainly occurred during the initial step of simulation where nodes are rapidly joining the system. However, as the system reaches the steady state in terms of available nodes and jobs, CAN maintenance overhead becomes stabilized which results in a very low overall cost for maintaining the entire CAN (as seen from Table 6.1).

Figure 6.4 shows the effects of randomizing job ownership in this large scale experiment. Compared to the experimental set with fewer nodes in the system (Figure 5.6), the distribution of job ownership becomes a little worse. However, this is not because of the larger scale of available nodes and jobs in the system but because of the smaller job inter-arrival time (from 4 seconds down to 200 milliseconds). Since the random walking along the T dimension is based on the aggregated information propagated periodically through virtual peers, the decision for a random owner might be made based on stale information as jobs are arriving with a high rate. However, we can still achieve a good distribution of job ownership since the algorithm does not depend on any distributions of node capabilities and job requirements.

6.3 Summary

In this chapter, we have performed a much larger scale experiment than in previous chapters to measure the ability of our system to scale gracefully as more nodes and jobs are injected into the system. By employing a comprehensive suite of load balancing techniques based on initial static pushing mechanism and supplemental dynamic load balancing scheme, our system could achieve good load balance with low matchmaking cost. Also, we could minimize the system load mainly imposed by maintaining the CAN space, by employing the modified heartbeat messaging scheme, which is not affected by the populations of available nodes and jobs in the system.

Chapter 7

Related Work

P2P 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 [50] 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 systems such as CAN [76], Chord [85], Pastry [78] and Tapestry [98] are also capable of scaling to very large numbers of machines and simultaneous requests for service. Based upon these basic P2P services, recently there have been several research efforts to combine P2P and Grid computing techniques to improve the robustness, reliability and scalability of commonly available client-server based desktop grid infrastructure [91].

7.1 Peer-to-Peer Systems

Peer-to-Peer systems can be defined as "distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority" [6]. Therefore, no node in the P2P system acts as a pure server or a pure client, as opposed to the traditional server-client model.

Existing P2P systems can be divided into two main categories based on the network structure: *structured* and *unstructured*. Structured P2P systems employ a deterministic structure to interconnect the peers and organize the file indexes, while in unstructured P2P systems each peer is randomly connected to a fixed number of other peers (often called neighbors) so that overlay network is created nondeterministically.

7.1.1 Unstructured P2P Systems

In unstructured P2P systems, each peer maintains a constant number of connections to other peers (neighbors), and the placement of data files is completely independent of the overlay topology. Due to the lack of an underlying deterministic structure in those systems, the prevailing resource location method is *flooding* the network by propagating queries in a breadth-first or depth-first manner until the target file is found. Clearly, flooding is not scalable since it creates a large volume of unnecessary traffic in the network. To limit the number of messages generated by flooding, each message is tagged with a *Time-To-Live* (TTL) field. The TTL indicates the number of hops away from its source a query can propagate so that each message is propagated into the network until the TTL expires. A small TTL value can reduce the network traffic, however the search may fail to find a target file although it is existing somewhere in the system. Many other techniques have been proposed to alleviate the excessive traffic problem caused by flooding and to deal with the traffic/coverage trade-off [92] such as random walks, multiple random walks, hybrid methods that combine flooding and random walks [40], directed searches based on statistical information [63], forwarding indexes [28], or the incorporation of semantic information [84, 97].

Unstructured P2P systems are generally more appropriate for accommodating

highly-transient node populations (high-churns). Examples of the popular unstructured systems are Gnutella [42], Kazaa [50], Publius [94], Edutella [70] and FreeHaven [30].

7.1.2 Structured P2P Systems

Structured P2P systems have emerged mainly in an attempt to address the scalability issues that unstructured P2P systems have. These systems are equipped with a distributed indexing service which is based on hashing so that they are often called *distributed hash table* (DHT) [9, 39, 49, 69, 76, 78, 85, 98]. Peers and files are mapped, usually through the same hash function, to a key space. Peers and file indexes are organized in a rigid structure according to their keys, which facilitates the location of files. Most structured P2P systems support a scalable solution for exact match queries in O(log N) hops, where N is the number of available nodes in the system. However they do not support directly keyword searches which constitute the core of queries in real P2P systems.

Structured P2P systems are more scalable than unstructured ones, in terms of traffic load, but need to have strong self-organization capabilities in order to maintain their deterministic structure. A main disadvantage of structured P2P systems is that it is difficult to maintain the structure required for an efficient

routing of messages with a very transient node population where nodes are joining and leaving the system with a high rate [24].

7.2 Unstructured P2P-based Matchmaking Mechanisms

Research such as [21, 45, 46] proposed a P2P architecture to locate and allocate resources in Grid environment employing a Time-To-Live (TTL) mechanism. TTL-based mechanisms are relatively simple but effective ways 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 (that meets the job constraints), even though such a resource exists somewhere in the network, because of the TTL mechanism (lack of *Completeness*).

Research such as [68, 74] adopt the *super-peer* model to design a P2P-based Grid information service. The super-peer model has been originally proposed to achieve a balance between the inherent efficiency of the centralized search and the autonomy, load balancing and fault-tolerant features offered by the distributed search [96]. A super-peer node acts as a centralized server for a number of reg-

ular peers (collecting information about the all nodes that it is managing), while super-peers connect to each other to form an overlay network that exploits P2P mechanisms at a higher level. Resource discovery is based on the TTL-like message forwarding among the super peers which is similar to the approach presented in Iamnitchi et al. [45, 46]

Similar to the resource discovery mechanisms based on the super peer model, Talia et al. [86] proposed a P2P architecture composing of two layers. The lower layer is a hierarchy of Index Services (provided by the Globus Toolkit [37, 41]) which publish information owned by each Virtual Organization (VO) [38]. The upper layer is the P2P layer which collects and distribute this information. The P2P layer includes two types of OGSA [38] compliant Web Services: Peer Services used to perform resource discovery and Contact Services that allow Peer Services to organize themselves in a P2P network. An extension of the Gnutella protocol is adopted to exchange matchmaking messages among the Peer Services in the P2P layer.

The CCOF (Cluster Computing on the Fly) project [62, 99] conducted a comprehensive study of generic searching methods (such as Centralized, Expanding Ring, Random Walk, Advertisement-based and Rendezvous Point searches) in a highly dynamic P2P environment to locate idle computer cycles throughout the

Internet. However, the host availability model in that work is not based on the resource requirements of the jobs nor the varying resource capabilities of nodes in the system (lack of *Expressiveness*).

Awan et al. [10] proposed a distributed cycle sharing system that utilizes a large number of participating nodes to achieve robustness through redundancy on top of an unstructured P2P network. By employing efficient uniform random sampling using random walks, probabilistic guarantees on the performance of the system could be achieved. The Organic Grid [23] proposed a self-organizing and fully decentralized approach to the organization of the computation. A large computational task is divided into small subtasks and each subtask is encapsulated into a mobile agent, which is then released on the Grid and finds appropriate resources based on autonomous behaviors of each agent. Balanced Overlay Networks (BON) [15] encode information about each node's available computational resources, resulting in a self-organized network that allows jobs to be assigned to free nodes via short random-walks. However, as for the CCOF project, the job allocation models in these work do not consider the resource constraints nor the varying resource capabilities in the system.

Marzolla et al. [66, 67] proposed another system for locating Grid resources by organizing system nodes as a tree-structured overlay network, where each node

maintains complete information about the set of resources it manages directly and a condensed description of the resources present in the sub-trees rooted in each of its neighboring nodes. However, all of these work are lack of fault-tolerance mechanism which is essential to improve the robustness and reliability of desktop grid computing systems.

Cappello et al. proposed a computational P2P system called XtremWeb [20] whose aim is to investigate the issues for turning a large scale distributed system into a parallel computer with classical user, administration and programming interfaces possibly harnessing 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 which is not scalable and robust as in the traditional desktop grid system.

7.3 Structured P2P-based Matchmaking Mechanisms

Butt et al. [16, 17] employed Pastry [78] to enable an efficient resource location in various computing environments. They allowed distributed Condor pools [33] to self-organize into a P2P overlay and locate nearby resource pools in the physical network for flocking [17]. Also they built a system that allows the sharing

of computational resources that builds on the Java properties of portability and safety, the credit system and scalable P2P networks [16]. However similar to the unstructured P2P-based approaches, they used a Time-To-Live (TTL) mechanism which lacks of *Completeness*.

Studies on encoding static or dynamic information about computational resources using a DHT hash function for resource discovery have also been conducted [7, 12, 18, 25, 43, 71]. Research such as [7, 12, 18, 43] employ one DHT per each resource attribute and perform matchmaking for the multi-attribute queries based on either controlled flooding [7] or sequential search [12, 18] which have shortcomings with respect to search performance with a large number of resource attributes (lack of *Low overhead*). Registering all resource attributes in a single DHT which enables an efficient matchmaking [25, 71] can have another problem in terms of load balancing. A small fraction of the nodes can contain a majority of the resource information whenever there are many nodes that have 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 *Parsimony*).

Heine et al. [44] used DHTs for storing semantic information in Grids. Peers

provide resource descriptions and background knowledge in ontology based on description logic and each peer can query the network for available resources. However, their work are lack of fault-tolerance mechanism which is essential to improve the robustness and reliability of desktop grid computing systems.

The WaveGrid [101] system constructed a "timezone-aware" overlay network based on Content-Addressable Network (CAN) [76] and utilized idle night-time cycles geographically distributed across the globe. However, the host availability model in this work is not based on the resource requirements of the jobs nor the varying resource capabilities of nodes in the system.

7.4 Dynamic Load Balancing Mechanisms

Dynamic load balancing concepts are widely used for distributing loads in locally distributed systems [83] or thread migration techniques [11]. Zhou et al. [100, 101] incorporated this concept by employing two distinct scheduling steps: *initial scheduling* and *later migration*. A client initially schedules its jobs on a host in the current night-time zone and when the host machine is no longer idle, the job is migrated to a new night-time zone. They are the first to investigate migration strategies in a peer-based desktop grid systems [100]. However, as we

described earlier, they do not allow users to specify resource requirements of their jobs. Therefore, it is much simpler model than our schemes where a node receiving a migrated job should be able to meet resource constraints of the job. The Condor system uses *preemptive resume scheduling*, which moves jobs before they complete (preemption) in order to meet the needs of system participants (such as owners, users and system administrators) or to deal with the inevitable heterogeneity of available computers [79]. However, Condor is based on a traditional centralized server-client architecture so that it has limited scalability and robustness compared to our decentralized job redistribution schemes.

Chapter 8

Conclusions and Future Work

In this chapter, I conclude this dissertation by reviewing the thesis and its contributions and present some directions for future work.

8.1 Thesis and Contributions

In this dissertation, I supported the following thesis: *decentralized resource management can be employed to create scalable desktop grid computing systems*. The goal of this work was to investigate the problem of building a scalable infrastructure for executing Grid applications on a widely distributed set of resources. Such infrastructure must be decentralized, robust, highly available, and scalable, while efficiently mapping application instances to available resources throughout the system. The main contributions made by this dissertation include:

1. **An efficient decentralized matchmaking framework**

   I designed and built a modified Content-Addressable Network (CAN) where each resource type corresponds to a distinct CAN dimension. To address the problems of one-to-one mapping of nodes to zones and jobs having very similar requirements, I augmented both job and node descriptions with a randomly assigned value in a virtual dimension. The virtual dimension ensures that all jobs and nodes are unique, and helps balance load even when the actual jobs and nodes are similar. Also, to support both of minimum and exact matches for the resource requirements of jobs, I integrated all types of resources in a single CAN consisting of multiple disjoint sub-CANs through virtual peers and 1-dimensional transformation. By leveraging such an architecture, incoming jobs specifying any types of resource requirements are efficiently matched with system nodes through proximity in an N-dimensional resource space without wasting any resources in the system.

   Our modified CAN has a different structure compared to the original CAN [76] where the coordinates in all dimensions are randomly generated. This is because our CAN is constructed based on semantic resource dimensions. Therefore, it is difficult for us to prove the matchmaking cost theoretically.

However, I supported this claim through experimental results obtained via simulations for both smaller and larger scale of node and job populations. The results show that as we increase the number of nodes and jobs by a factor of 10, the average matchmaking cost increased only 30%.

2. **Comprehensive decentralized load balancing mechanisms**

   The load on individual nodes in a desktop grid consists of application load (the jobs to be executed), and system load (load imposed by the workings of the underlying system). Here, the load balancing means that we distribute the application load across multiple candidate nodes in the system that can run the given jobs (i.e., meet the resource requirements specified by those jobs). Comprehensive means that we employ both static and dynamic load balancing schemes to improve overall system throughput and user response time.

   I have designed comprehensive decentralized load balancing mechanisms that can greatly improve the quality of load balancing and obtain very competitive performance even compared to the idealized centralized scheme. Overall steps of load balancing algorithms are composed of initial and static load balancing based on job pushing mechanism and supplement it with

lightweight and effective dynamic load balancing mechanisms that can redistribute the jobs if needed. By employing such a framework, our system can prepare and handle any source of potential load imbalance that can be caused by the heterogeneity of the system nodes and jobs and stale load information in the decentralized P2P desktop grid.

3. **A set of optimizations to reduce the system load**

   A scalable P2P desktop grid computing system must be able to not only balance the load of job executions but also introduce minimal and fair overheads among the nodes in the system. Unfortunately, non-uniform distributions of jobs and nodes can cause the system to distribute the system load unevenly across nodes. This system load can come from either monitoring job executions or maintaining overall CAN space and it can limit the scalability of our system. This overloaded system maintenance cost is not sustainable in the P2P desktop grid since every node in our system is a peer so that unfair distribution of system loads cannot attract the participation of desktop machines.

   The majority of the system load comes from maintaining the CAN by individual nodes independently and periodically sending heartbeat messages

to neighboring nodes in CAN. By employing partial updates mechanism, the size of each update message is limited to d * 2 (for both directions) * *PU_Threshold* * *SN* where d is the number of dimensions and *SN* denotes the size of a single neighbor information. A single neighbor information includes zone, coordinates and neighbors of neighbors (indirect neighbors) information, etc. *PU_Threshold* denotes the number of maximum neighbors information for each direction of a dimension included in a single update message. Also, by employing round-robin heartbeat messaging scheme, the number of messages sent per period is limited to the 2 * d so that every node in our CAN sends the same number of messages as the original CAN [76]. These mechanisms ensure that the system load can be reduced and distributed more fairly among available system nodes.

8.2 Future Work

We foresee many possible extensions to the work presented in this dissertation. Although I provided a set of algorithms and techniques that can build a scalable P2P desktop grid system, many improvements can still be made and explored.

Investigating the Real System Until now, I have mainly implemented and evaluated my decentralized matchmaking and load balancing mechanisms in an event-driven simulator. This is to see the behaviors of our system under various scenarios of node capabilities and resource requirements of jobs. Our project team has developed a real prototype system based on CAN and is testing the system with real workloads provided by our astronomy collaborators in the University of Maryland. While we are switching from simulator to the real system, we confront many other challenges that could not be addressed in the simulation environment. Therefore, we will investigate the behaviors of our real prototype system and report the results of executing real applications in the near future.

Incorporating Multi-Processor (Core) Nodes In my thesis work, I have assumed that when a job is assigned to a run node, the run node processes only one job at a time and in FIFO order. However, as hardware technologies evolve and the needs for computing power to solve complex scientific applications increases, we have to be able to run *multiple independent* jobs at a single physical node in the system. For example, if a node has multiple processors or multiple cores, allowing only a single job execution in that node is a waste of those resources. This kind of highly capable machines is becoming more prevalent in the desktop

grid computing environment as multi-core desktop machines are introduced. Furthermore, we can even consider running *multi-threaded* jobs in the P2P desktop grid system if we can find a node containing multiple processors (cores) that can match the required number of threads for that job. This brings us new challenges since we have to devise a mechanism to represent highly capable nodes (having more than one processing units) in the CAN space and also the matchmaking algorithm should be able to consider the number of required processors for a single (multi-threaded) job.

Executing Dependent or Parallel Jobs In our current formulation of the problem, there are no dependencies between jobs, but if computational scientists also use the system for data analysis of results, then the system will have to distinguish between job types (simulation vs. analysis) and perform the jobs in the correct order (analysis after simulation of a given problem), and make the output of a simulation job available as the input for the corresponding analysis job(s). Whenever jobs are dependent on each other, there should be a scheduler such as Condor's DAGMan [88] which can manage and monitor the overall execution flows of jobs in the system.

Also, we can consider running *parallel* jobs on the pool of machines in the

P2P desktop grid. To run the parallel jobs in the P2P desktop grid system, we have to find multiple nodes that can run these jobs and also preferably they should be close enough each other in terms of network distance. This means that now our P2P desktop grid system should be able to consider building an ad-hoc cluster that can be constructed from the pool of machines on the fly and also incorporate other hardware specifications such as network bandwidth and latency.

Long-Term Plans In the near future, small embedded computing devices with wireless network will become more prevalent. These devices will enable the automated and remote control of living environments and the transparent use of the Internet. This phenomenon is already happening as more smartphones such as Apple iPhone [8] or BlackBerry [14] are introduced into the market and become more and more popular. As these devices multiply, enormous amount of information will be generated and carried by such small mobile devices. This means that the concept of a "resource" will not be limited to the conventional desktop computers or workstations any more and will evolve to include all kinds of *computable and connectible devices*. This post-desktop model of computing in which information processing is integrated into everyday devices and activities is often called ubiquitous computing or pervasive computing. In such ubiquitous com-

puting environments, I believe that the techniques for harnessing and managing those resources will become more important. I intend to develop efficient, robust, and reliable resource management schemes in such dynamic pervasive computing environments. To achieve this goal, I believe that many challenging questions should be addressed and my research experience during Ph.D work will be the foundation of identifying and devising solutions to such questions.

Glossary

- **Categorical Resource Type** is such as a specific type of operating system or processor. Categorical resource constraints require a singular value for that resource (exact match) (described in detail in Section 3.2).

- **Centralized Matchmaker** is an online scheduling mechanism 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 (described in detail in Section 3.1.2).

- **Continuous Resource Type** is such as memory or disk size, or CPU speed. For continuous resource types, matchmaking requires that a node meet or exceed a job's requirements (minimum match) (described in detail in Section 3.2).

- **Distributed Hash Table** (DHT) are a class of decentralized distributed systems that provide a lookup service similar to a hash table: (name, value) pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given name. Responsibility for maintaining the mapping from names to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption.

- **Dynamic Load Balancing** addresses problems that arise from our static load balancing mechanisms for assigning jobs to nodes that can arise for various reasons, including the heterogeneity of the available nodes or the jobs to be run, and from stale information in the P2P system. Dynamic load balancing scheme redistributes the jobs if needed based on either pulling jobs to lightly loaded node or pushing jobs away from heavily loaded nodes (described in detail in Section 4.2).

- **Heartbeat Messages** are soft-state messages for failure recovery in our system. There are two different types of heartbeat messages: messages between owner and run nodes and messages between each node and its neighbors. Former messages are used for failure recovery whenever either owner

or run node fails or depart the system. Latter messages are used for maintaining the CAN DHT (described in detail in Section 2.1 and 5.1).

- **Injection Node** is the node where a job insertion is initiated from. The injection node can be any arbitrary node in the system and DHTs provide an external mechanism that can find an existing node in the system (described in detail in Section 2.1).

- **Load Balancing** is the process of distributing application load (the jobs to be executed) across multiple candidate nodes in the system that can run the given jobs (i.e., meet the resource requirements specified by those jobs)

- **Matchmaking** is the process of matching jobs with physical resources, and consists of finding an appropriate node for running a job based on the resource constraints in the job profile and the current (distributed) state of the nodes in the system.

- **Modified Heartbeat Messaging** addresses the heartbeat message exchanging problem between virtual peers and their neighbors to improve the scalability of our system. This is because a virtual peer maintains a large number of neighbors in the sub-CANs so that it can send not only a larger number of messages but also the size of each message becomes large. Modified

Heartbeat Messaging is composed of two different mechanisms: partial update mechanism and round-robin heartbeat messaging (described in detail in Section 5.1).

- **Owner Node** is originally the node where a job is routed to based on the coordinates of the job. After introducing the concept of randomized ownership (Section 5.2), an arbitrary node in the system can become the owner node of a job. However, we can still find the owner node of a job deterministically based on the (pseudo-)coordinates of the job. The owner node is responsible for monitoring job executions (described in detail in Section 2.1 and 5.2).

- **Partial Update Mechanism** is to reduce the size of a single update message. Whenever a node sends information about its neighbors, it may only send partial neighbor information. For this purpose, we use a threshold value, PU_Threshold, which limits the number of neighbors that are included in a periodic update message in each direction (upper or lower in each dimension). Therefore, even with only partial information about neighbors, each node will let its neighbors know about at least one and at most PU_Threshold neighbors in each direction (we select the PU_Threshold neigh-

bors from each direction randomly) (described in detail in Section 5.1).

- **Pushing Mechanism** is an improved static load balancing scheme. Nodes periodically send load information towards the origin in each CAN dimension. This information is aggregated at each step, resulting in each node having partial information about load in all regions of the CAN space containing nodes more capable,– exactly those nodes that are also able to run the node's jobs. In times of high load, a node can therefore push jobs towards regions of high capability and low load, based solely on local information (described in detail in Section 4.1).

- **Pseudo Coordinates** of a job are randomly generated and used only for determining the owner and "real" coordinates (corresponding to the job's resource requirements) are used for matchmaking process (described in detail in Section 5.2).

- **Random Walking along *T* dimension** is based on aggregated information propagated through virtual peers and used for determining the owner node of a job randomly across all available nodes in the system (described in detail in Section 5.2).

- **Rendezvous Node Tree** (RNT) is a distributed data structure built on top of an underlying DHT, which in our implementation is Chord [85]. 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 (described in detail in Section 3.1.2).

- **Round-robin Heartbeat Messaging** is to reduce the number of update messages sent by a virtual peer. The virtual peer limits how often it sends heartbeat messages to any given neighbor through the partial update mechanism described previously, which only lengthens the average time between heartbeat messages sent to each neighbor (described in detail in Section 5.1).

- **Run Node** is the node that processes job executions. The run node should be able to meet the resource requirements specified by the jobs (described in detail in Section 2.1).

- **Specialized Routing in the *T* Dimension** addresses the problem of routing bottleneck across multiple sub-CANs. Whenever a physical peer tries to route a request to the virtual peer, it sends the request to one of the neighbors of the virtual peer (rather than sending directly to the manager of the virtual

peer). This prevents all routing requests delivered from the one sub-space to another sub-space from always going through the virtual peers (described in detail in Section 3.2.3).

- ***T* Dimension** is the transformed dimension of all categorical resource dimensions. For this purpose, we use a locality-preserving Space Filling Curve, specifically the Hilbert Space Filling Curve (HSFC) [59, 80, 81] (described in detail in Section 3.2.1).

- **Virtual Dimension** is an additional dimension in a CAN space where the coordinates are generated uniformly at random. The virtual dimension ensures that all jobs and nodes are unique, and helps balance load even when the actual jobs and nodes are similar (described in detail in Section 3.1).

- **Virtual Peers** are used for covering unoccupied sub-spaces in CAN. To integrate categorical resource types, we divide the CAN space into multiple disjoint sub-spaces where in each sub-space all categorical resource types are exactly the same and address the connectivity issue through virtual peers (described in detail in Section 3.2).

- **Virtual Peer Manager Node** is the node where a virtual peer is mapped to and maintains all information about the virtual peer (e.g., neighbor list) and

processes any routing requests for its assigned virtual peer(s) (described in detail in Section 3.2.2).

BIBLIOGRAPHY

[1] The Great Internet Mersenne Prime Search. Available at *http://www.mersenne.org/prime.htm*.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Dec. 1999.

[3] 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.

[4] D. P. Anderson, C. Christensen, and B. Allen. Designing a Runtime System for Volunteer Computing. In *Proceedings of the 2006 IEEE/ACM SC06 Conference*, Nov. 2006.

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

[6] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[7] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of the 2nd International Conference on Peer-to-Peer Computing*, Sept. 2002.

[8] Apple. Available at http://www.apple.com/.

[9] M. S. Artigas, P. G. Lopez, J. P. Ahullo, and A. G. Skarmeta. Cyclone: a Novel Design Schema for Hierarchical DHTs. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005)*, Aug. 2005.

[10] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Unstructured Peer-to-Peer Networks for Sharing Processor Cycles. *Parallel Computing*, 32(2), Feb. 2006.

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

[12] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the ACM SIGCOMM*, Aug. 2004.

[13] K. Bhatia. Peer-To-Peer Requirements On The Open Grid Services Architecture Framework. *GFD-I.049 OGSAP2P Research Group, Global Grid Forum*, July 2005.

[14] BlackBerry. Available at http://www.blackberry.com/.

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

[16] 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 Virtual Machines Research and Technology Symposium (VM'04)*, May 2004.

[17] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. In *Proceedings of the 2003 IEEE/ACM SC03 Conference*, Nov. 2003.

[18] M. Cai, M. Frank, J. Chen, and 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)*, Nov. 2003.

[19] B. Calder, A. A. Chien, J. Wang, and D. Yang. The Entropia Virtual Machine for Desktop Grids. In *Proceedings of the 1st ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

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

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

[22] A. J. Chakravarti, G. Baumgartner, and M. Lauria. Application-Specific Scheduling for the Organic Grid. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Nov. 2004.

[23] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(3), May 2005.

[24] Y. Chawathe, S. Ratnasamy, and L. Breslau. Making Gnutella-like P2P Systems Scalable. In *Proceedings of the ACM SIGCOMM 2003*, Aug. 2003.

[25] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer Discovery of Computational Resources for Grid Applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID 2005)*, Nov. 2005.

[26] 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.

[27] N. Coleman, R. Raman, M. Livny, and M. Solomon. Distributed Policy Management and Comprehension with Classified Advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, Apr. 2003.

[28] A. Crespo and H. Garcia-Molina. Routing Indices For Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, July 2002.

[29] A. Demers, , D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, Aug. 1987.

[30] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[31] Dutch University Backbone. The Distributed ASCI Supercomputer 2 (DAS-2). Available at http://www.cs.vu.nl/das2, 2006.

[32] EGEE Team. LCG. Available at http://lcg.web.cern.ch/LCG, 2004.

[33] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12(1):53–65, May 1996.

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

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

[36] 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.

[37] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

[38] I. Foster and C. Kesselman, editors. *The GRID 2: Blueprint for a New Computing Infrastructure*. Elsevier / Morgan Kaufmann, 2004.

[39] M. J. Freedman, E. Freudenthal, and D. Mazi. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'2004)*, Mar. 2004.

[40] C. Gkantsidis, M. Mihail, and A. Saberi. Hybrid Search Schemes for Unstructured Peer-to-Peer Networks. In *Proceedings of the IEEE Infocom 2005*, Mar. 2005.

[41] GlobusToolkit. Available at http://www.globus.org/toolkit/.

[42] Gnutella. Available at http://www.gnutella.com/.

[43] R. Gupta, V. Sekhri, and A. K. Somani. CompuP2P: An Architecture for Internet Computing using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1306–1320, Nov. 2006.

[44] F. Heine, M. Hovestadt, and O. Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004)*, Nov. 2004.

[45] A. Iamnitchi and I. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Proceedings of the 2nd IEEE/ACM International Workshop on Grid Computing (GRID 2001)*, Nov. 2001.

[46] A. Iamnitchi and I. Foster. A Peer-to-Peer Approach to Resource Location in Grid Environments. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Re-*

source Management: State of the Art and Future Trends, pages 413–429. Kluwer Academic Publishers, 2004.

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

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

[49] M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

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

[51] J.-S. Kim, B. Bhattacharjee, P. J. Keleher, and A. Sussman. Matching Jobs to Resources in Distributed Desktop Grid Environments. Technical Report CS-TR-4791 and UMIACS-TR-2006-15, University of Maryland, Department of Computer Science and UMIACS, Apr. 2006.

[52] 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.

[53] J.-S. Kim, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman. Using Content-Addressable Networks for Load Balancing in Desktop Grids (Extended

Version). Technical Report CS-TR-4863 and UMIACS-TR-2007-16, University of Maryland, Department of Computer Science and UMIACS, Mar. 2007.

[54] 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.

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

[56] 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.

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

[58] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *Proceedings of the 18th*

International Parallel & Distributed Processing Symposium, Apr. 2004.

[59] J. Lawder. Calculation of Mappings Between One and n-dimensional Values Using the Hilbert Space-filling Curve. Technical Report BBKCS-00-01, Birkbeck College, Aug. 2000.

[60] J. Ledlie, J. Schneidman, M. Seltzer, and J. Huth. Scooped, Again. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[61] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[62] V. Lo, D. Zhou, D. Zappala, Y. Lin, and 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)*, Feb. 2004.

[63] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proceedings of the 16th ACM International Conference on Supercomputing (ICS 2002)*, June 2002.

[64] 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.

[65] M. Marsh, J.-S. Kim, B. Nam, J. Lee, S. Ratanasanya, B. Bhattacharjee, P. Keleher, D. Richardson, D. Wellnitz, and A. Sussman. Matchmaking and Implementation

Issues for a P2P Desktop Grid. In *2008 National Science Foundation Next Generation Software Workshop (NSFNGS 2008)*, Apr. 2008.

[66] M. Marzolla, M. Mordacchini, and S. Orlando. Resource Discovery in a Dynamic Grid Environment. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, Sept. 2005.

[67] M. Marzolla, M. Mordacchini, and S. Orlando. Peer-to-peer systems for discovering resources in a dynamic grid. *Parallel Computing*, 33(4-5):339–358, 2007.

[68] C. Mastroianni, D. Talia, and O. Verta. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. In *Proceedings of the European Grid Conference (EGC2005)*, Feb. 2005.

[69] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002.

[70] W. Nejdl, B. Wolf, C. Qu, S. Decker, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[71] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[72] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A Common Data Management Infrastructure for Adaptive Algorithms for PDE Solutions. In *Proceedings of the 1997 IEEE/ACM SC'97 Conference*, Nov. 1997.

[73] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. Request for Comments 1546, Internet Engineering Task Force, Nov. 1993.

[74] D. Puppin, S. Moncelli, R. Baraglia, N. Tonellotto, and F. Silvestri. A Grid Information Service Based on Peer-to-Peer. In *Proceedings of the Euro-Par 2005*, Aug. 2005.

[75] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

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

[77] 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.

[78] 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)*, Nov. 2001.

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

[80] H. Sagan. Space-Filling Curves. *Springer-Verlag*, 1994.

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

[82] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[83] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, 1992.

[84] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In *Proceedings of the IEEE INFOCOM 2003*, Mar. 2003.

[85] 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 the ACM SIGCOMM*, Aug. 2001.

[86] D. Talia and P. Trunfio. Peer-to-Peer Protocols and Grid Services for Resource Discovery on Grids. *Grid Computing: The New Frontier of High Performance Computing*, 14:83–103, 2005.

[87] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *Proceedings of the ACM SIGCOMM*, Aug. 2003.

[88] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, A. J. Hey, and G. Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11, pages 299–336. John Wiley, 2003.

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

[90] The TeraGrid Project. Npaci. Available at http://www.teragrid.org, Mar. 2006.

[91] 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.

[92] D. Tsoumakos and N. Roussopoulos. A Comparison of Peer-to-Peer Search Methods. In *Proceedings of the International Workshop on the Web and Databases (WebDB 2003)*, June 2003.

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

[94] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

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

[96] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, Mar. 2003.

[97] D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. Exploiting Locality for Scalable Information Retrieval in Peer-to-Peer Networks. *Information Systems*, 30(4):277–298, 2005.

[98] 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.

[99] D. Zhou and 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*, Apr. 2004.

[100] D. Zhou and V. Lo. Wave Scheduler: Scheduling for Faster Turnaround Time in Peer-Based Desktop Grid Systems. In *Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing*, 2005.

[101] D. Zhou and V. Lo. WaveGrid: a Scalable Fast-turnaround Heterogeneous Peer-based Desktop Grid System. In *Proceedings of the 20th International Parallel & Distributed Processing Symposium*, Apr. 2006.