Unstructured Peer-to-Peer Networks for Sharing Processor Cycles [1]

Asad Awan [*], Ronaldo A. Ferreira, Suresh Jagannathan,
Ananth Grama

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA
E-mail:{awan,rf,suresh,ayg}@cs.purdue.edu

Abstract

Motivated by the needs and success of projects such as SETI@home and genome@home, we propose an architecture for a sustainable large-scale peer-to-peer environment for distributed cycle sharing among Internet hosts. Such networks are characterized by highly dynamic state due to high arrival and departure rates. This makes it difficult to build and maintain structured networks and to use state-based resource allocation techniques. We build our system to work in an environment similar to current file-sharing networks such as Gnutella and Freenet. In doing so, we are able to leverage vast network resources while providing resilience to random failures, low network overhead, and an open architecture for resource brokering. This paper describes the underlying analytical and algorithmic substrates based on randomization for job distribution, replication, monitoring, aggregation and oblivious resource sharing and communication between participating hosts. We support our claims of robustness and scalability analytically with high probabilistic guarantees. Our algorithms do not introduce any state dependencies, and hence are resilient to dynamic node arrivals, departures, and failures. We support all analytical claims with a detailed simulation-based evaluation of our distributed framework.

Key words: unstructured peer-to-peer networks, distributed randomized algorithms, distributed cycle sharing, load balancing

---

[*] Corresponding Author.

1 Introduction

The use of a large number of unreliable hosts over a wide-area network to solve compute-intensive problems has been pioneered by projects such as SETI@home [1], genome@home [2], and distributed.net [3], among others. While key advantages such as increased performance, reliability, and scalability motivate the use of decentralized peer-to-peer (P2P) systems as opposed to traditionally used client-server models, we consider the broader goal of an open environment for cycle sharing as a major driver for a move towards a P2P solution for harnessing idle CPU cycles of Internet hosts. Such an environment would allow any participant to submit tasks, in contrast to the inflexible nature of a client-server model. Furthermore, an open P2P system provides an incentive for peers to contribute their resources, expecting cycles in return, as opposed to the altruistic basis for current systems.

In this paper, we propose and evaluate an unstructured P2P architecture for distributed cycle sharing among Internet hosts. The dynamic nature of P2P networks resulting from high arrival and departure rates motivates our choice of an unstructured (stateless) model. A majority of Internet-deployed examples of successful, resilient, large-scale, and massively-distributed systems rely on such unstructured topologies (e.g., Gnutella [4], Freenet, etc. for file sharing). Our design decision trades off the overheads of building and maintaining a structured overlay (with associated guarantees on network path lengths) for a less expensive model (with probabilistic guarantees on network delays).

Our P2P based architecture bears features that make it particularly suitable for constructing an open infrastructure for distributed cycle sharing. Specifically, it does not require an organization to manage and deploy a dedicated infrastructure. In contrast, one may consider the deployment of BOINC [2] based projects – each project deploys a server complex, including a directory server, and participating clients register with the project using a master URL designated for the project. In our architecture, nodes, which could assume the role of a project owner or a working participant, join the P2P network and know only about their immediate neighbors. Jobs are identified by unique identifiers, rather than network node addresses. All information [3] about participants and jobs are stored in a distributed fashion among peer nodes. This avoids a single point of failure and improves scalability. Project owners submit jobs to the network and do not need to store the identity of nodes computing the tasks. This allows use of a large number of nodes in the network without introducing state management overhead on the project servers. Computed results are routed to the project owners using content based addressing rather than

---

[2] Berkeley Open Infrastructure for Network Computing (BOINC) [5] is a software platform for developing distributed computing applications.
[3] The integrity of this data is preserved using standard cryptographic techniques. This is discussed later in the paper.

network node address. We present protocols that efficiently enable these functionalities in unstructured P2P networks.

Our distributed cycle sharing system utilizes the large number of participating nodes to achieve robustness through redundancy. Due to the scale of the system and the volatile nature of the participants, we rely on randomized algorithms to build our protocols. The use of efficient randomized algorithms affords simplicity, while preserving probabilistic guarantees on the performance of the system. The underlying approach is novel and carries its share of complexities, solutions to which form the key contributions of this paper. The design goals of our system are:

(1) Low job makespans by ensuring load balance.
(2) Resilience to node failures and frequent node arrivals and departures.
(3) Validation of computed output by redundant distributed computations.
(4) An interface for monitoring job progress and performance evaluation.
(5) An accounting framework for the resources contributed by various nodes.

The key substrate supporting our design is efficient uniform random sampling using random walks. Uniform sampling in networks provides the basis for a variety of randomized algorithms and is of interest on its own as well. In context of our system, uniform sampling allows us to design randomized algorithms for load balancing, applying redundancy to support fault tolerance, and building a probabilistic rendezvous service for monitoring task progress and contributions of participating nodes.

Uniform Sampling in Unstructured Networks

Uniform sampling in a network requires randomly selecting a node, such that every node in the network has the same probability of being selected. A trivial approach to this problem would be to collect the entire set of node identifiers at each node and index randomly into this set of identifiers. This simple approach, however, does not work for our target applications because the overhead of frequently updating system state at each node (if at all possible) would be prohibitive. An alternate approach to this problem relies on the notion of a random walk. Starting from an initial node, a random walk (of predetermined length) transitions through a sequence of intermediate nodes with probabilities defined for each link and ends at a destination node. The likelihood of terminating a random walk at any node determines whether the walk is a uniform sampling random walk or not. Formally, we define a uniform sampling random walk as follows:

Definition 1.1 (Uniform sampling using random walk) *A random walk of a given length samples uniformly at random from a set of nodes of a connected network if and only if the walk terminates at any node i belonging to the network, with probability $1/N$, where N is the number of nodes in the network.*

The key parameters of interest in sampling via random walk are: (i) a uniform sample should be generated irrespective of the topology of the network, and (ii) the length of the walk required to reach stationarity (mixing time of the walk) should be small. A number of researchers have studied properties of random walks, e.g., Lovasz [6] provides an excellent survey of these techniques. The simplest random walk algorithm selects an outgoing edge at every node with equal probability, e.g., if a node has degree four, each of the edges is traversed with a probability 0.25. It can be shown that the probability distribution associated with target nodes becomes stationary after a finite length random walk (also known as the mixing time for the corresponding Markov chain). This length can be shown to approach $O(\log N)$. These concepts are discussed in greater detail in Section 4. The main drawback of the simple random walk is that, while it reaches a stationary distribution, this distribution is not uniform for typical networks. In fact, it can be shown that the probability of terminating a random walk at a node is directly proportional to the degree of the node. In the context of conventional unstructured P2P networks, where node degrees can vary significantly, this does not correspond to an acceptable uniform sample.

Much like other typical applications of random walks, our system is sensitive to the quality of uniform sampling. Biases in sampling may result in poor performance of randomized algorithms, congestion in underlying networks, and significant load imbalances. Thus, realizing random walks that yield uniform sampling irrespective of topology is a key focus of our work. In addition to the quality of uniform sampling, an important performance parameter is the length of the random walk. Since longer random walks generally correspond to a larger number of network messages, it is highly desirable to minimize the length of the walk.

Technical Contributions

The paper makes the following specific contributions:

- It presents a scalable, robust, and efficient architecture for a P2P resource-sharing network.
- The basis for the proposed network is a load balancing, replication, and monitoring scheme that relies on efficient randomized algorithms. It presents a random walk based algorithm for uniform sampling in large real-world networks with low overhead. This sampling methodology provides a substrate for our randomized algorithms.
- It provides empirical results that demonstrate the efficiency of our algorithms for computing a large number of tasks on unstructured P2P networks with high node failure and arrival rates. For example, we show that our randomized algorithm based P2P infrastructure achieves an efficiency of over 40% compared to an ideal parallel ensemble, even when network overhead is significant.

The rest of this paper is organized as follows. In Section 2, we summarize related results. In Section 3, we present an overview of our randomization-based P2P computing architecture. In Section 4, we show how uniform sampling can be achieved via random walks. We also present an algorithm that allows efficient (short length) random walks to obtain uniform sampling. In Section 5, we empirically evaluate the performance of our architecture. We show that our architecture yields high efficiencies for distributed computations. We also evaluate strategies for job replication. We derive conclusion from our work in Section 6.

2   Related Work

SETI@home [1], genome@home [2], and distributed.net [3] are among the early examples of distributed cycle sharing systems that utilize a large number of Internet hosts. However, these systems are custom made for executing tasks originating at a single source. In contrast, our system allows sharing CPU cycles among peers and running multiple jobs from disparate users in the network. Projects such as BOINC [5] and XtremWeb [7] also aim to provide a multi-application global (Internet-wide) computing platform. However, both of these projects are based on a hierarchical client/server architecture and rely on a setup initiated and maintained by a single organization. On the other hand, we architect our global computing platform based on a decentralized unstructured peer-to-peer network comprised of Internet hosts.

The Condor [8] project aims at utilizing distributed computing resources in a network to provide high throughput. A mechanism called ClassAd is used to advertise attributes of available resources and jobs. Condor acts as a broker between the resources and the jobs, using the attributes provided to it. Similar to our system, this provides an open environment in which multiple users can submit jobs. However, task management in Condor is centralized, which makes the environment more tightly coupled. It is assumed that Condor will be deployed and managed by an organization. In contrast our architecture allows self-organization of participants. Instead of using state information we rely on randomization. Furthermore, our system can provide a decentralized ClassAd based task allocation mechanism using the rendezvous service and hence can be considered complementary to Condor. Similarly, an implementation using our architecture could easily borrow mechanisms such as checkpointing and sandboxed execution from Condor.

Our work can also be considered complementary to much of the work on grid computing including Globus [9,10], Legion [11], Avaki [12], Purdue University Network Computing Hub (PUNCH) [13], and Sun's Grid Engine [14]. Each of these systems implements a centralized or a hierarchical management component, which is different from our fully decentralized approach. Our P2P communication fabric and randomized techniques can be applied to these systems as well.

Butt et al. [15] uses a well known structured P2P network (Pastry [16,17]) for locating and allocating computing resources. A Java VM is used to execute and monitor the progress of the execution on peers. A credit system for accounting services is also provided. In contrast, we build our system on top of an unstructured P2P network, motivated by the success of massive unstructured networks for file sharing. Our main emphasis is on architecting allocation and communication mechanisms that yield high efficiency and are robust in the face of high node departures and failures. Our system also provides mechanisms for job monitoring, aggregation, reputation, and communication between oblivious hosts. We use randomization techniques that provide probabilistic guarantees and have low overhead. The architecture presented in [18] is another example of CPU sharing using a structured P2P network (Chord [19]). Similarly, G2-P2P [20] also relies on a node addressing scheme based on Pastry [16] and Tapestry [21].

3 Architectural Overview

In this section, we provide a brief overview of unstructured P2P networks and describe a simple randomized job allocation scheme that achieves good load balance. We also motivate the need for redundancy in the context of target applications and show how our protocol caters to replicating tasks in the network. A key aspect of our P2P cycle sharing environment is a decentralized *rendezvous service* for monitoring job progress, supporting loosely coupled inter-task communication, and aggregating completed tasks. We describe the distributed construction and the probabilistic guarantees on the performance of this service. We also show how our architecture can be leveraged to manage reputation of participating hosts in the network as a means to counter "free-riders" in the system.

3.1 Unstructured Peer-to-Peer Networks

Unstructured P2P networks are characterized by decentralized control and lack of a well-defined overlay topology. Node connections are formed randomly. However, as the graph evolves, long-lived nodes gain a higher number of connections due to their higher availability and as they become known, over time, by other peers in the network. High degree nodes are called super nodes. The emergent self-organized network graph has a highly skewed degree distribution with few nodes having high degrees while most nodes have only few neighbors and are often connected to the super nodes. These networks have some desirable features such as low diameter, and resilience to random failures and frequent node arrival and departures. More importantly, they are simple to implement and incur virtually no overhead in topology maintenance. Consequently, many real-world large-scale peer-to-peer networks are unstructured. However, the lack of structure makes it difficult to locate

shared resources in the network. In such networks, the naive method for locating resources is by flooding a (hop) limited neighborhood. This approach, however, has high overhead in terms of network messages and does not provide guarantees of locating existing objects in the network.

In this paper, we build our system on top of an unstructured network, and present algorithms that provide performance guarantees with high probability (w.h.p.)[4] while incurring low overheads.

3.2   Job Allocation with Redundancy

We first present a simple job allocation strategy that achieves appropriate redundancy and good load balance. Conventional unstructured P2P networks comprise of tens, even hundreds of thousands of nodes. Consequently, computational resources exist for building sufficient redundancy into the system. There are two main motivating factors for redundancy:

(1) Resilience. In an open Internet environment, failures (and departures) of nodes must be expected. In such an environment, replication of the same task to multiple hosts is needed to account for failure of nodes.
(2) Validation. We can expect that some of the nodes would return wrong results, either because of malicious or other reasons. Results from several nodes can be cross-checked to detect faults, and possibly select a correct output (e.g., simply using majority) from the available set of reported results. Indeed, several current systems such as SETI@home use similar methods.

We assume that a job, J, can be broken down into n independent[5] subtasks. We denote by N the number of Internet hosts in the peer-to-peer network. Let p be the number of Internet hosts that are engaged in computing the job J. The subtasks of J can be clustered into batches $(b_1, b_2, \ldots)$, each with K subtasks. We discuss, later in this section, the possible considerations to be taken while choosing K. During job submission, each batch is replicated by a factor $r \geq 1$. For example, $r = 2$ implies that two nodes will be assigned the same batch to compute.

A simple randomized job submission algorithm that allows replication can be constructed as follows:

(1) A host A that wants to submit subtasks of its job sets the batch size K and a replication factor r. Each batch has an associated meta-information which

---

[4] The term w.h.p. indicates probability of success equal to $1 - \frac{1}{N^{\Omega(1)}}$.
[5] We subsequently discuss the need for inter-subtask communication and show how our architecture addresses this requirement.

includes the job identifier and a unique name (created as a hash, e.g., MD5, of the batch content). The meta-information always accompanies the batch.

(2) For each batch, host A selects a node, B, uniformly at random by performing a random walk and submits a batch to it. The replication factor r is also sent with the batch.

(3) Each node that receives a batch decrements r by one and if $r > 0$ sends a copy of the batch to another node chosen uniformly at random. The updated value of r is sent with the batch. The address of node B (i.e., the direct descendent of node A) is also sent. Thus, each batch is replicated at a total of r nodes in the network.

Steps 2 and 3 are atomic, i.e., node A knows that either they were successfully and entirely completed or that they failed. This is implemented using the 2-phase commit strategy as follows. Each time a node receives a batch, it sends an ACK to the node that sent it the batch (i.e., its parent). Furthermore, each time a node successfully receives an ACK from its child, it sends a message to (its respective) node B. After a timeout period, node B compares the number of such messages received, r', with $r - 1$ and resubmits batches to make up for the difference. This is done using the same protocol as step 3, except this time $r - 1 - r'$ represents the number of replicas needed. Once a total of $r - 1$ messages are received it sends an activation message [6] to all the $r - 1$ nodes and itself. A message is sent to node A indicating successful distribution of the given batch. Each of the involved nodes (including node A) monitor the life of node B and abort this transaction if node B fails. As r is usually a small constant number, and the messages exchanged are only control messages, the associated overhead of this scheme on node B does not result in a bottleneck for node B. The state information, involved in this mechanism, can be discarded once the transaction completes successfully. Observe that the replication is done by a node B and not node A. We use this approach to avoid overload on node A when the number of subtasks is large.

In ideal conditions, note that the number of batches sent by A remains n/K since redundant replication is taken care of by nodes downstream. Similarly, note that replication of each batch occurs in parallel. The total number of messages for submitting the batches in the network is given by nr/K. In real-world situations, this translates to reducing the cost of job submissions for master nodes. Certainly, any node failures during this transaction result in higher overheads due to resubmissions.

Several jobs can be initiated on the network concurrently. Since our job allocation protocol is based on randomization, a node may be assigned more than one batch, either of the same job or of different jobs. A node processes batches one at a time on a first come first served basis. We emphasize that this randomized protocol is

---

[6] This message does not require an ACK, i.e., the batch will not be resubmitted to cater to the failure of a node which successfully received a batch earlier but failed before activation.

simple, does not maintain any persistent state information, and has a low overhead arising from random walks for uniform sampling.

3.2.1 Uniform Sampling and Load Balancing

Assume that a total of m batches need to be assigned to the N processors. If we assign batches uniformly at random to the N processors, we can provide bounds on the quality of load balance achieved. Given m batches, we answer the following questions:

(1) What is the expected fraction of nodes that will have a batch assigned to them?
(2) What is the probability that a node gets a given number of jobs to perform?
(3) What is the maximum load on a node?

The arguments we present next suggest that using $m = N \log N$ provides good utilization of the network w.h.p. and at the same time yields a low probability of high load imbalance.

Lemma 3.1 *Given m batches, the expected fraction of processors that have a batch assigned to them is $1 - e^{-m/N}$.*

Proof: The probability that a given processor does not get a batch, when a single batch is submitted, is $1 - 1/N$. Thus, when m batches are distributed, the probability that a given processor does not get any of these batches is given by:

$$(1 - \frac{1}{N})^m \approx e^{-m/N},$$

assuming that N is large. We define an indicator random variable X_i, which assumes the value 1 if the ith host did not get any batch, and 0 otherwise. We want to find the expected number of processors that do not get any batch. Using linearity of expectation, we have:

$$E[X] = E\left[\sum_{i=}^{N} X_i\right] = \sum_{i=1}^{N} E[X_i] = N(1 - \frac{1}{N})^m \approx Ne^{-m/N}$$

Thus, the number of nodes that will get at least one batch to process is $N - Ne^{-m/N}$, and the fraction of nodes that will get at least one batch to process is $(N - Ne^{-m/N})/N = 1 - e^{-m/N}$. □

This lemma simply implies that if $m \leq N$ at most, approximately, 65% of the nodes will be used. Similarly, if $m \geq N \log N$, with high probability all nodes will be used. This result is significant because the utilization level of the system is an important performance metric for a parallel ensemble.

Next, we determine what load balance characteristics can be claimed if job submission is based on uniform sampling. We find the probability that a given node gets *exactly M* batches to process, when a total of m batches are submitted to the distributed system consisting of N nodes. Note that these M batches can be any combination chosen from the m batches. Then, the probability is given by:

$$\binom{m}{M}\left(\frac{1}{N}\right)^M\left(1-\frac{1}{N}\right)^{m-M}$$

Given that m and N are large compared to M, the above expression can be approximated by:

$$\binom{m}{M}\left(\frac{1}{N}\right)^M\left(1-\frac{1}{N}\right)^{m-M} \approx \frac{e^{-m/N}(m/N)^M}{M!}$$

Similarly, the probability that a node gets *at least M* batches is given by:

$$\binom{m}{M}\left(\frac{1}{N}\right)^M$$

Building on the above derivations, we show in the following lemma, that a high load on a single processor occurs with low probability. We use the case where the number of batches $m = O(N\log N)$, which refers to the condition that yields good system utilization (i.e., almost all N nodes get a batch to process) with high probability.

Lemma 3.2 *If $m = O(N\log N)$, the probability that the maximum load on any node is more than $\Omega(\log N)$ is low.*

Proof: For simplicity, we set $m = N\log N$. We denote the maximum load (i.e., the maximum number of batches assigned to any node) to be M. The probability that *any* node is assigned at least M batches is given by the following expression.

$$N\binom{N\log N}{M}(\frac{1}{N})^M$$

We can derive a bound on this probability by setting $M = e^2\log N$ (i.e., $M = \Omega(\log N)$). Thus, the probability that any node has a load more than $e^2\log N$ can be shown to be low (i.e., $1/N$) as follows.

$$N\binom{N\log N}{M}(\frac{1}{N})^M \leq N(\frac{eN\log N}{e^2\log N})^M(\frac{1}{N})^M$$
$$= \frac{N}{e^{e^2\log N}} = \frac{N}{N^{e^2}} \leq \frac{1}{N}.$$

$\square$

Using this lemma, we show in the following theorem, that job submission using uniform random sampling results in a good load balance amongst the nodes in the network.

Theorem 3.1 *When $m \geq N \log N$ the load imbalance on a node diminishes.*

Proof: When $m = N \log N$, a perfect deterministic algorithm allocates $\log N$ batches to each machine. For this case, from the previous lemma, we know that the probability that the maximum load on a machine is $\Omega(\log N)$ is low. Therefore, a randomized algorithm that uses uniform sampling to distribute tasks approaches a deterministic algorithm and yields good load balance. □

3.2.2 Choosing Batch Sizes

Selecting an appropriate batch size in a distributed environment is a challenging task and depends on several variables. Kruskal and Weiss [22] shows that when the running times of the subtasks are independent and identically distributed (i.i.d.) random variables with mean μ and variance σ^2, then the estimated completion time is given by:

$$E(T) = \frac{n}{p}\mu + \frac{nh}{pK} + \sigma\sqrt{2K\log p} \tag{1}$$

where n is the number of subtasks, p is the number of processors, h is the network delay, and K is the batch size. It is assumed that $K/\log p$ is large, and for smaller values the error is not substantial. This expression is quite general and holds for processing time distributions including exponential, gamma, Weibull, uniform, deterministic, and truncated normal distributions. The variance in the time required to complete the processing of a task, σ, depends on the following parameters:

(1) The number of concurrent jobs scheduled on the processors. This corresponds to the number of subtasks allocated to a single processor, and depends on the load balance achieved by the job submission algorithm.
(2) The processing capability of participating hosts.
(3) The variation and non-deterministic nature of processing requirements of each subtask.

Each term in Equation 1 has important implications. The first term corresponds to the time it would take an ideal parallel system with p processors to compute the n subtasks. The second terms captures the benefit from aggregating jobs in terms of reducing the communication cost. The final term represents the overhead due to uneven finishing times of the processors. The most important implication of this result is the tradeoff between communication, which decreases as $1/K$, and the variance in processing times, which increases as $\sqrt{K}$.

In the above discussion, there is no mention of the failure (and departure) of nodes in the network. This, however, needs to be considered because there is an important

relationship between the size of the batch and the lifetime of a node. As a simple illustration, consider the minimum processing time of all batches, $\min T(b_i)$. If $\min T(b_i)$ is greater than the expected lifetime of the nodes, the system would be reduced to using only a fraction of the nodes whose lifetime is large compared to the expected completion time of the job. Heavy load on such nodes implies that it would take much longer to complete the jobs, which further reduces the set of nodes that have suitable lifetimes. Furthermore, being able to use long lived nodes would require a job allocation scheme that maintains state information. This can be prohibitively expensive in large dynamic networks. Our use of a randomized approach avoids these overheads, but at the same time is susceptible to failure under the conditions just discussed. Thus, a batch size should be small enough, so that the required time for processing the batch is comparable to the lifetime of the nodes. Lifetime of nodes is also important in the context of developing a replication strategy, as discussed in Section 3.2.3.

In summary, the size of a batch, K, should be:

(1) Large enough so that the network overhead is reduced,
(2) Small enough so that total job completion is minimally affected by variations in processing times taken by hosts, σ, and
(3) Small enough so that the computation time required for a given host is comparable to the host's lifetime.

Furthermore, the results from Section 3.2.1 suggest that if the total number of tasks to be computed in the network (from all jobs) is n', then the number of batches, n'/N, should be $N \log N$ for good load balance and network utilization. We propose $K = \log^2 N$ as an ideal compromise for aggregating tasks. This allows computation of a very large number of tasks in the network, $n' = N \log^3 N$. It also results in a good network utilization since the number of batches is sufficient. For example, for a network of 100,000 hosts, around 150 million tasks can be executing, which would achieve high utilization, while the load imbalance would be bounded. As K grows faster than $\log N$, Equation 1 can approximate (with low error) the total running time for n' tasks. Plugging in these values shows that this value of K gives a low network overhead as $n'h/pK = h \log N$. It also results in a low impact on execution time variations as $\sigma \sqrt{K \log p} = \sigma log^{3/2} N$.

3.2.3 Multi-Step Replication

We revisit the replication strategy keeping in view the fact that increased replication at submission time implies increased time to completion of the jobs. Increased makespan of a job implies that more nodes would leave the network during that time, conditioned on the distribution of lifetime of the nodes in the network.

Let γ be the number of nodes leaving the network in unit time. We denote the time for completion of n tasks by $T(n)$. As defined earlier, r denotes the number of

replicas of a given task in the network. Recall that the protocol given earlier in this section performs replication at job submission time. Then, $n_f = \gamma T(nr)$ gives the number of nodes leaving the network during the makespan of the job. The level of replication to deterministically counter the failures would require $r = n_f$. This, in turn, asserts the stability condition: $\gamma = n_f/T(n \cdot n_f) \leq 1/T(n)$, i.e., at most one node failure over the job makespan. This stability requirement is degenerate.

Thus, a job submission protocol should use a multi-step replication strategy instead of replication-at-initiation. This algorithm is well suited for a high node failure rate environment. The protocol works as follows:

(1) A host, A, that wishes to submit subtasks of its job, sets the batch size, K. The replication factor r is set to one.
(2) For each batch, host A selects a node uniformly at random, by performing a random walk, and submits a batch to it.
(3) Host A also calculates T, which is the time it would take an ideal parallel ensemble to complete the job, as $T = n/N\mu$. It then waits for a time εT, where ε is the expected efficiency of our distributed system without failures. This can be estimated using Kruskal's equation (cf. Equation 1).
(4) After waiting for this time period, it collects the jobs that have been computed using the algorithm presented in Section 3.3.2.
(5) Host A determines the jobs that did not run to completion and resubmits them with $r = 2$. If n_f jobs were missing, T is calculated again as $\max(r \cdot n_f/N, 1) \cdot \mu$. The waiting and re-submission, with r incremented at each step, continues until results for all tasks are retrieved.

Note that this scheme does not require the complex multi-party atomic transaction for replication, as required by the replication-at-initiation protocol, described in Section 3.2. However, the overhead of replication in terms of bandwidth usage is experienced by the node that introduces the job to the network. In Section 5, we show using simulation that this multi-step protocol in fact executes tasks much faster than the replication-at-initiation strategy when this strategy uses more than 3 replicas. However, the key advantage is that this algorithm achieves 100% job completion, which the other algorithm cannot achieve, in most cases.

3.3 Rendezvous Service

The rendezvous service provides a communication fabric between nodes that are oblivious to each other. The key used for communication is a resource identifier or a resource query string, rather than a node address. This is analogous to the directory service in a client-server architecture. However, unlike its client-server based counterpart, a rendezvous service does not have a centralized repository and the peers need not know or register with any predefined server node. The required

information is maintained in a completely distributed fashion among the peers.

3.3.1 Construction

Nodes become a part of the rendezvous service by creating a "rendezvous service set" (RS-set). The RS-set of each node contains pointers to $\sqrt{N \log N}$ peers selected uniformly at random. This construction is similar to the probabilistic quorum presented in [23]. The creation of the RS-set at each node occurs in a distributed fashion asynchronously. Each node is responsible for maintaining $\sqrt{N \log N}$ live peers in its RS-set, when there are node failures. The provider of a resource publishes its resource identifier to its RS-set. Similarly, the node looking for this resource sends a query to its own RS-set. If there is an intersection between RS-sets of the producer and the consumer then the consumer can access the resource.

Theorem 3.2 *Any two RS-sets of size $\sqrt{N \log N}$ nodes intersect w.h.p.*

Proof: Since the RS-sets contain $\sqrt{N \log N}$ nodes selected uniformly at random, the probability that a given node of one RS-set is not in the other RS-set is given by $1 - \frac{\sqrt{N \log N}}{N}$. Thus, the probability that none of $\sqrt{N \log N}$ nodes of one RS-set are in the other RS-set is:

$$\left(1 - \frac{\sqrt{N \log N}}{N}\right)^{\sqrt{N \log N}} \approx \frac{1}{N}.$$

The probability that at least one node in the two sets intersect is $1 - \frac{1}{N}$, which implies that the two RS-sets intersect with high probability. □

Note that this service has very low overhead. The overhead of creating the RS-set is amortized over the life of the node. Similarly, only $\sqrt{N \log N}$ messages are required for each query. We can compare this with flooding, where the number of network messages increases exponentially with the number of hops. In the rest of this section, we show how this service is used in our architecture.

3.3.2 Monitoring and Merging Jobs

When a node completes the execution of a batch, it informs its RS-set using the job identifier. With high probability, one of the nodes in the RS-set of this node also belongs to the RS-set of the owner of the job. Such an intersecting node retrieves the results. Each of the nodes in the RS-set of job owners maintain independent bit vectors, where the jobs received by them are marked. The owner requests these vectors from its RS-set, once in a given interval. It may also download the results for the completed tasks. Alternatively, the owner may ask its RS-set nodes to submit completed jobs to it once a certain number of new results are available. If replication factor $r > 1$ is used in submitting jobs the owner would get multiple copies of the same job. This information can be used to verify that the results match, and thus

provide a validation mechanism.

3.3.3 Reputation Monitor

Reputation monitoring requires publishing the information (ID) of malicious nodes to the RS-set of the node that discovers the malicious activity. The mechanisms for detecting malicious activity are independent of this subsystem and an example is provided later. Repeated offenses of a malicious node result in several nodes reporting against the node. If node x wants to query the reputation of node y, it simply queries its RS-set. The overlap between the RS-sets reveals the required information. Using this information, and observing that several nodes reported against y, node x may not accept y in its RS-set, or not submit or accept jobs from node y. This discourages free-riders and malicious users in the system. The integrity of the reputation monitor is robust to collaborative malicious activity since a large number of randomly chosen nodes (RS-sets of multiple nodes) keep this information. Such systems have been proposed in the context of conventional file-sharing P2P networks and have been shown to handle free-riders in a scalable manner [24].

As a simple example of discovering malicious activity, consider the case where a node submits wrong results. The RS-set of the master receives results computed by all participating nodes. It compares the multiple results for the same batch and accepts the most repeated result as the correct result. The ID of nodes which submitted wrong result is broadcast to the RS-set of the job owner.

3.3.4 Decoupled Communication

A node might need to search for information without knowing which node has the required information. For example, a node processing a task may need to know the result of some preceding task. Such cases may arise if the subtasks of a job are not completely independent. Similarly, several other examples of resource location can be cited. In these scenarios, a node needs to send a query about the requested resource to its RS-set and with high probability it will intersect with the RS-set of the provider of the resource.

3.4 Security Considerations

Using a decentralized approach for submitting jobs and aggregating results makes the system prone to several security threats. We discuss some of these issues here and describe how they can be handled in our architecture.

Attack on load balancing: a malicious node may try to overload another node by selecting it repeatedly during the random sampling process. Our architecture is in-

herently safe from such attacks because a node under attack can delegate the task to another randomly chosen node without affecting the load balancing scheme. This is because the load balancing scheme only requires that the nodes are chosen randomly and the above action does not affect the "randomness" of uniform sampling (cf. Section 4).

Modification of data passing through intermediate nodes: in the protocols described earlier in this section, data destined for a given node may be forwarded via another node. This happens when the RS-set nodes forward results to the owner of the job and in the replication-at-initiation protocol. In these cases, standard digital signatures suffice to ensure the integrity of the data. BOINC [5] provides an example of the application of these techniques. Similarly, in the replication-at-initiation protocol the first node in the chain of replication is responsible to make sure that r replicas are sent to random nodes. Using digital signatures the job owner can be sure that r replicas were forwarded, however, it cannot be ensured that all the r nodes are not collaborating malicious nodes.

Several other scenarios involving threats due to collaborative malicious actions need to be evaluated in greater detail. Similarly, some of the unresolved security issues of existing systems, e.g., [25], are pertinent to our project. A comprehensive evaluation of the security vulnerabilities and feasible counter measures is a key focus of our ongoing and future work in the area of open infrastructures for Internet-scale cycle sharing.

4   Uniform Sampling With Random Walks

In this section we introduce random walks and show how they can be used to perform random sampling. If the underlying network does not have a regular degree distribution, i.e., if few nodes are connected to many, and many nodes are connected to a few nodes, then a random walk, with transitions from a node to its randomly chosen neighbor, does not yield a uniform sample. We revisit Kruskal's equation and argue that skewed sampling results in bad load balance and long job makespan. We show how the transition probability from one neighbor to another must be modified to achieve a uniform sampling, and give an algorithm that computes the required transition probabilities.

4.1   Sampling With Random Walks

Random walks can be abstracted as Markov chains defined over a state space and a given state transition matrix. The network nodes form the state space and the probability of moving from a node to its neighbor govern the transitions. Using a

Markov chain model, we show in this section that (1) a random walk of a given minimum length on a connected aperiodic graph (which represents the network) reaches a stationary node sampling distribution, and (2) a *simple random walk* cannot achieve uniform sampling unless nodes in the network have identical numbers of connections. We also discuss various parameters that determine the length of the random walk required to achieve a stationary sample distribution.

Let $G(V,E)$ be a simple connected undirected graph representing a distributed system with $|V| = N$ nodes and $|E| = e$ links. The degree, or number of links, of a node i, $1 \leq i \leq N$, is given by d_i. The set of neighbors of a node i is given by $\Gamma(i)$, where edge $(i,j) \in E, \forall j \in \Gamma(i)$. The $N \times N$ adjacency matrix of G is given by $A = \{a_{ij}\}$, where $1 \leq i,j \leq N$, $a_{ij} = 1$ if the edge $(i,j) \in E$, and 0 otherwise. The corresponding $N \times N$ transition probability matrix, given by $P = \{p_{ij}\}$, is the probability of moving from node i to a node j in one hop. P is a row-stochastic matrix, i.e., $\sum_j p_{ij} = 1$.

For a *simple random walk* the transition from node i to its neighbor is governed by the transition probability matrix P, where $\forall j \in \Gamma(i)$, $p_{ij} = 1/d_i$ and 0 otherwise. The sequence of nodes can be denoted as $\{X_t, X_{t+1}, ...\}$, where $X_t = i$ implies that at step t the walk is at node i.

If we consider nodes in G as states in a finite state space, then the random walk represents a discrete-time stochastic process, $\{X_t\}_{t \geq 0}$. For this stochastic process we have,

$$Pr(X_{t+1} = j | X_0 = i_0, ..., X_{t-1} = i_{t-1}, X_t = i)$$
$$= Pr(X_{t+1} = j | X_t = i) = p_{ij} \tag{2}$$

Equation (2) simply implies that a random walk is *memoryless*, i.e., during a random walk the probability of transition from node i to node j in one step depends only on node i. Thus, a random walk can be conveniently modeled as a Markov chain, more specifically a homogeneous Markov chain, since the right hand side of Equation (2) is independent of t. Such a Markov chain has the following properties: it is irreducible if the graph G is connected and is aperiodic if G is aperiodic. A graph G is aperiodic if the greatest common divisor of the length of all cycles in the graph is 1. In particular, an undirected aperiodic graph cannot be bipartite, which is a reasonable assumption for real networks in which connections are established randomly.

4.1.1 Convergence to Random Sampling

It is well known that an irreducible and aperiodic Markov chain has a stationary distribution $\pi^T = \pi^T P$, and $\pi^T = \pi^T P^t$ follows (where P^t implies t-step transitions). It is easy to show ([26], page 132) that π_i, the component corresponding to node i, $1 \leq i \leq n$, is $d_i/2e$. From $\pi^T = \pi^T P$, we see that π is a left eigenvector of P

with eigenvalue [7] 1. The right eigenvector for eigenvalue 1 is $\mathbf{1}$ (a vector of all ones), since $P\mathbf{1} = \mathbf{1}$. It follows that $P^\infty = \mathbf{1}\pi^T$. This implies that a very long walk converges to the stationary distribution π irrespective of the initial distribution, i.e., the starting point of the walk.

The above results indicate that a long enough random walk converges to a random sample irrespective of where the walk started. Thus, a random walk is a good candidate for random sampling in a network. However, we also know that the resulting sample distribution is dependent on the degree of the node: $\pi_i = d_i/2e$. This last result implies that the random sample is uniform ($\pi_{uniform} = (1/N)\mathbf{1}$) only if the graph G is regular (i.e., the degrees of all nodes are equal). Since typical large scale real-world unstructured networks tend to have non-uniform degree distributions (e.g., power-law degree distribution of unstructured P2P networks [28]), uniform sampling in practical scenarios poses a significant challenge.

4.1.2   Length of Walk for Random Sampling

The sample distribution at step t of the walk depends on P^t, which in turn depends on the eigenstructure of P. From the Perron-Frobenius theorem, we have $P^t = \lambda_1^t v_1 u_1^T + O(t^{m_2-1}|\lambda_2|^t)$, where v_1 is the right eigenvector corresponding to eigenvalue λ_1 and u_1 is the left eigenvector, and m_2 is the algebraic multiplicity of λ_2 (see, [27] Chapter 6). Rewriting the above equation, we have $P^t = P^\infty + O(t^{m_2-1}|\lambda_2|^t)$. These results simply imply that

$$P^t = \mathbf{1}\pi^T + O(t^{m_2-1}|\lambda_2|^t). \tag{3}$$

As $|\lambda_2| < 1$, when t is large, $|\lambda_2|^t \approx 0$. Therefore, the smaller the second largest eigenvalue modulus (SLEM), the faster the convergence to stationary distribution. As a result, a walk of smaller length is required for random sampling. Also note that there is a limit on the minimum length on the walk, i.e., any walk of length greater than this minimum yields a random sample. The length of the required walk, or the *mixing time*, is often approximated as $O(\log N)$[6], however the exact factors involved depend on the construction (and thus the SLEM) of the transition probability matrix.

4.2   Uniform Sampling in Nonuniform Networks

As mentioned in Section 4.1.1, a random walk of a given minimum length converges to a stationary distribution π. If the stationary distribution $\pi_{uniform}$ is such

---

[7] Since P is a non-negative primitive $N \times N$ matrix (i.e., irreducible and aperiodic), from basic linear algebra, we also know that P has N distinct eigenvalues $1 = \lambda_1 > |\lambda_2| \geq \cdots \geq |\lambda_N|$ [27].

that $\pi_{uniform} = (1/N)\mathbf{1}$, the random walk will terminate at any node in the network with equal probability (c.f. Definition 1.1).

However, if the stationary distribution is $\pi_i = d_i/2e$, we pick the high degree nodes with a much higher probability. This implies that such nodes will have a high number of tasks to compute. The variance in the node sampling increases the variance of the processing time of the nodes, as the processing resource of the node would be divided over the tasks assigned to it. If the resulting variance of the running time from the mean processing time of the nodes is s^2, then the Kruskal's equation would be written as:

$$E(T) = \frac{n}{p}\mu + \frac{nh}{pK} + \sqrt{\sigma^2 + s^2}\sqrt{2K\log p} \tag{4}$$

This directly impacts the expected running times of jobs. Note that variance of the degree of the nodes may be very high if the underlying graph follows power-law degree distribution. In Section 5.2.1, we experimentally demonstrate that using simple random walks for sampling in nonuniform networks yield a poor load balance, and consequently a large job turnover time.

4.2.1 Modifying Transition Probabilities

To achieve a uniform stationary distribution in an irregular graph, we need to modify its probability transition matrix.

Let P be a probability transition matrix of a Markov chain, then $\pi_{uniform}^T = \pi_{uniform}^T P$, which is the same as $(1/N)\mathbf{1}^T = (1/N)\mathbf{1}^T P$. This means that the sum of each column vector of P is 1, i.e., P is *column stochastic*. A probability transition matrix which is column stochastic in addition to being row stochastic is called *doubly stochastic*. Note that symmetric transition probability matrix are doubly stochastic. Thus, if we create a matrix with $p_{ij} = p_{ji}$ we will achieve a uniform stationary distribution, and hence a random walk using these transition probabilities will yield a uniform sample.

Two well known algorithms, maximum-degree algorithm (MD) [29] and Metropolis-Hastings algorithm [30,31], yield a symmetric transition probability matrix. However, these algorithms need a long walk to reach stationarity, if the graph has a highly skewed degree distribution. In our previous work [29], we present a detailed discussion and experimental evaluation of these algorithm, and suggested a new algorithm for building a transition matrix. Here, we reproduce the algorithm, however, the details and evaluation are omitted.

4.2.2 Random Weight Distribution Algorithm

In this section, we present our distributed algorithm, referred to as the Random Weight Distribution (RWD) algorithm. RWD is a completely decentralized algorithm that sets up transition probabilities in a connected network to enable efficient uniform sampling via random walks.

The algorithm proceeds as follows. In the initialization phase each node, locally, sets transitions probability as:

$$
p_{ij}^{rwd} = \begin{cases} 1/\rho & \text{if } i \neq j \text{ and } j \in \Gamma(i), \text{ where } \rho \geq d_{max} \\ 1 - d_i/\rho & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}
$$

Here, ρ is a static system parameter with the constraint that it should be greater than maximum degree, d_{max}. This parameter is static because we can sufficiently overestimate d_{max} knowing system properties (e.g., popular P2P clients have a maximum connection limit [32]). Note that this phase results in a high self-transition probability for low degree node. Also note that the resulting transition probability matrix is symmetric.

After the initialization is complete, each node attempts to distribute its self-transition probability randomly and symmetrically to its neighbors. The term *weight of a node* refers to the self-transition probability of the node at any given time during the execution of the algorithm. At a node i, the algorithm terminates when either the weight of the node becomes zero or the weight of all nodes $j \in \Gamma(i)$ becomes zero. Intuitively, a low self-transition probability implies that the walk mixes faster and converges to a stationary uniform distribution with a fewer number of steps. The pseudo code for the complete RWD algorithm is shown in Figure 1.

Remark 4.1 *Each step in the RWD algorithm maintains symmetry in the global transition probability matrix P^{rwd}. Therefore, the transition probability matrix remains symmetric when the algorithm terminates. Thus, a random walk based on P^{rwd} will have stationary distribution $\pi_{uniform}$.*

The overhead of messages due to our algorithm are minimal as explored in our earlier work [29].

5 Performance Evaluation

We present here detailed simulation results for various performance aspects of our system. First, we evaluate the efficiency of our system for different job loads. We

At each node i:

▷ *Initialization*
1.　　　$N := \Gamma(i)$
2.　　　$\delta := \texttt{Quantum}$
3.　　　$p_{ii} = 1 - d_i/\rho$
4.　　　**foreach** $j \in \Gamma(i)$ **repeat**
5.　　　　　$p_{ij} = 1/\rho$
6.　　　**end foreach**

▷ *Random Weight Distribution*
1.　　　**while** $p_{ii} \geq \delta$ **and** $N \neq \{\varnothing\}$
2.　　　　　$j := random(N)$
3.　　　　　$reply := send\_mesg(j, INCREASE)$
4.　　　　**if** $reply = ACK$ **then**
5.　　　　　　$p_{ij} := p_{ij} + \delta$
6.　　　　　　$p_{ii} := p_{ii} - \delta$
7.　　　　**else**
8.　　　　　　$N := N - j$
9.　　　　**end if**
10.　　**end while**

▷ *Receive Message Handler*
1.　　　$mesg := receive()$
2.　　　$j := get\_sender(mesg)$
3.　　　$type := get\_type(mesg)$
2.　　　**if** $p_{ii} \geq \delta$ **and** $type = INCREASE$ **then**
3.　　　　　$p_{ij} := p_{ij} + \delta$
4.　　　　　$p_{ii} := p_{ii} - \delta$
5.　　　　　$reply := ACK$
6.　　　**else**
7.　　　　　$reply := NACK$
8.　　　**end if**

Fig. 1. The Random Weight Distribution algorithm.

show that job allocation using random walks with transition matrix generated using the RWD algorithm yields good load balance due to uniform sampling. In comparison, job allocation using sampling based on simple random walk yields a highly skewed load balance. A comparison of the efficiency of the two schemes reflects the impact of sampling techniques. Next, we study the performance of our architecture under varying node failure rates. We compare the replication-at-initiation and the multi-step replication schemes in terms of job completion time and their resilience to failures. We show that replication-at-initiation is not able to fully recover results for tasks when failure rates are high, even when the replication factor is increased. Furthermore, increased replication results in significantly higher job completion times. In contrast, the multi-step scheme is always able to recover computed re-

sults for all of the submitted tasks and the overhead associated is low, compared to replication-at-initiation using a high replication level.

The main contribution of this experimental study is that it serves as a proof-of-concept of the feasibility of a randomization-based unstructured P2P computing environment.

5.1 Setup

Our simulation testbed implements the RWD algorithm for uniform sampling of nodes, the two job distribution protocols discussed in Section 3.2, and the rendezvous service for job aggregation and progress monitoring. These algorithms are implemented to function on top of an unstructured P2P network. The P2P network is a connected graph of nodes, where each edge denotes a link between the nodes. Hence, each node knows only about its neighbors in the graph. We use a power-law random topology for the network. Such graphs are often used in the literature to model large non-uniform network topologies. It is believed [33] that unstructured P2P networks have power-law topologies. In a power-law random graph, if the nodes are sorted in descending order of degree, the ith node has degree D/i^a, where D is a constant. The parameter $a = 0.8$ is used. This value of a is commonly used in evaluation studies of P2P networks [34]. The underlying topology is constructed by first selecting the degree of each node using a power-law distribution and then connecting them randomly. Motivated by real-world systems [32], we limit the maximum degree to 100. In typical P2P clients such as Limewire [32], these restrictions are often specified to restrict the number of connections of a given node in order to limit the load on the node. We fix the network size to $N = 10,000$ nodes. Base on this network size, the optimal number of subtasks (cf. Section 3) is equal to 7.8 million. We use this number in most of our experiments, unless otherwise stated.

5.1.1 Network Delays and CPU Time

Communication due to random walks, job submission, result aggregation, and progress monitoring has an associated network delay cost that affects the performance of our distributed cycle sharing ensemble. In our evaluation, we use network delay measurements taken from the Internet using thousands of node pairs on PlanetLab [35], as discussed below.

The P2P network is an overlay network, i.e., neighbors in the P2P network may not be neighbors in the underlying Internet graph. A message between any two nodes (even if they are neighbors) in the P2P network may actually be routed over several hops in the underlying network and the end-to-end delay between these nodes need to be considered. Furthermore, in the case of random walks, which consists

of several hops in the P2P network, we have to accumulate the per-hop delays. To realistically account for the network delays, we measured the end-to-end cost of transferring messages, using TCP, between thousands of node pairs on PlanetLab. We gathered delay samples for over $14,000$ pairs of nodes. The number of samples we could gather is limited by the availability of nodes on PlanetLab. PlanetLab consists of over 500 nodes but several of them could not be contacted during our experiment runs. The delays measured depend on the sizes of the messages transferred. For random walk messages, we used a size of 1KB. For a message containing a job to be executed, we used a maximum size of 500KB [8]. This size corresponds to the maximum work unit file of SETI@home [1]. As the P2P network used in our experiments consists of $10,000$ nodes, we require 100 million pairs, which is not feasible to collect using PlanetLab. Thus, we assign network delays (for each message size) between the $10,000$ nodes by sampling from the $14,000$ measurement set.

The next parameter to be configured for the experiments is the CPU time consumed per task. We assume that the CPU time per task is an exponential random variable with mean μ. The nature of computation to be executed determines μ, and this value varies greatly across different application domains. If the CPU time is large compared to the network costs then the efficiency of the system, in comparison to an ideal parallel ensemble, is higher. On the other hand, if the CPU time is smaller compared to the network costs then the efficiency of the system is lower. To allow a conservative evaluation of our system, we set $\mu = 11s$. This value corresponds to the mean network delay (from our Internet measurements) for transferring a 500KB file over TCP. Also, note that this choice favors using a higher number of subtasks per batch as per Kruskal's equation (cf. Equation 1).

5.2   Efficiency and Computational Throughput

We study the efficiency of our system in comparison to an ideal N processor parallel ensemble, which computes n tasks in $\mu n/N$ time. As stated in the discussion in Section 3, if $N \log N$ batches are submitted, then w.h.p. each processor has a task to perform. Similarly, if possible, we would like to have batch sizes approaching $K = \log^2 N$ tasks (which translates to an optimal $n = N \log^3 N$). In the cases where $n/K < N$, we use $K = max(n/N, 1)$. To evaluate the efficiency of our system for different job sizes and batch sizes, we use the parameters described in Table 1. Note that although we would like to have $N \log N$ batches each time, due to substantial network overhead, we prioritize larger batch size over the number of batches that can be submitted. As a result, when there are only 10,000 batches, $\approx 30\%$ of the machines in the network have no tasks to process. This is consistent with the

---

[8]   The exact size depends on the number of subtasks in the message. 500KB corresponds to the maximum number of subtasks (84 as discussed latter) in the batch.

predictions from Section 3.2.1. The resulting efficiency of the system is plotted in

Number of jobs, n		Cluster size, K		Number of batches	
N	$= 10000$	N/N	$= 1$	N	$= 10000$
$N \log N$	$= 92000$	$\log N$	$= 9$	N	$= 10223$
$N \log^2 N$	$= 846400$	$\log^2 N$	$= 84$	N	$= 10072$
$N \log^3 N$	$= 7800000$	$\log^2 N$	$= 84$	$N \log N$	$= 92858$

Table 1
Number of jobs and batch size parameters used for evaluating the efficiency of our system.

Figure 2. As predicted, the system has the highest efficiency when $K = \log^2 N$ and the number of batches is $N \log N$. The efficiency achieved in this case is 44.4%, which is excellent for such loosely coupled dynamic environments, given that the order of processing time is almost the same as the network overhead. We expect that in the real system there would usually be enough job submission requests to meet the optimum value of number of tasks in the system.
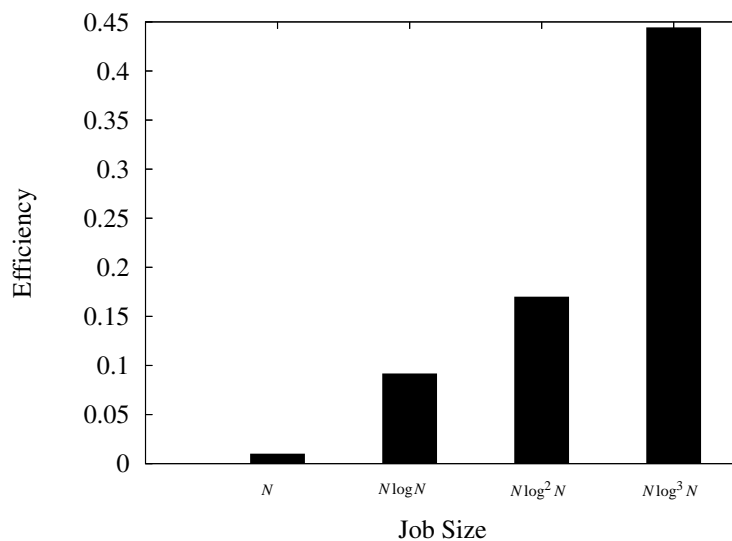

Fig. 2. Efficiency of our system when compared with an ideal parallel ensemble.

5.2.1 Effect of Sampling Techniques

Uniform node sampling is the underlying substrate for all our randomized algorithms. Our RWD algorithm computes transition probabilities in such a way that a random walk yields a uniform sample. In comparison, a simple random walk (SRW) is biased towards high degree nodes. We compare the load balance achieved using these two strategies and evaluate the impact of load imbalance due to the SRW algorithm.

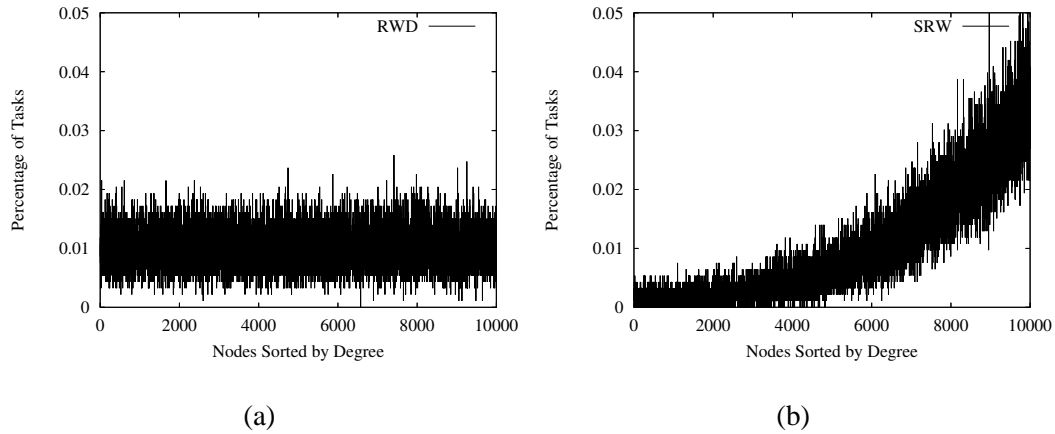To compare the load balance of the two schemes, we use the optimal parameters

Fig. 3. Load per node using uniform sampling with RWD (a) versus sampling using simple random walk (b).

$n = N \log^3 N$ and $K = \log^2 N$. The resulting percentage of the total number of tasks assigned to each machine are plotted in Figure 3. The *x*-axis of the plot represents nodes sorted in ascending order by degree. The number of tasks assigned to each machine for the sampling using transition matrix generated from RWD has a uniform distribution, with low load imbalance as seen in the plot in Figure 3(a). On the other hand the number of tasks assigned to a machine by sampling using SRW is biased to the degree of the node as seen in Figure 3(b). Thus, some nodes end up receiving almost 5 times higher load than other nodes.
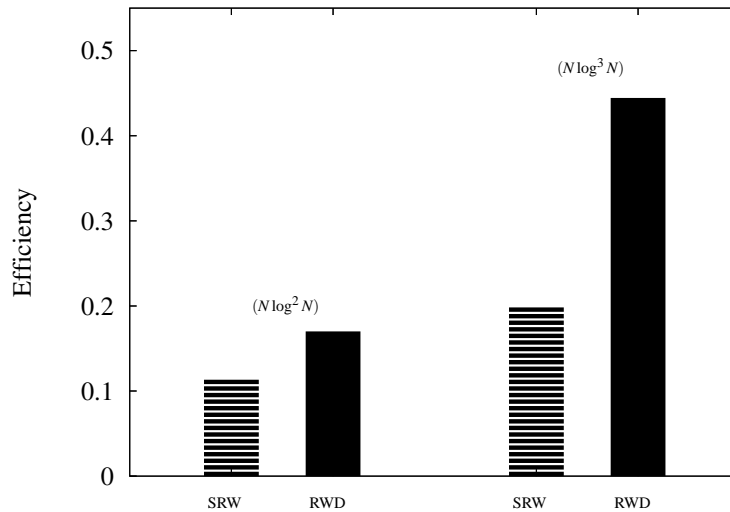

Fig. 4. Performance advantage of uniform sampling using random walk with RWD versus simple random walk.

A load imbalance implies that the job would take longer time to finish. We compare the system efficiency when batches are allocated with sampling using the two techniques. The number of jobs used are $n = N \log^3 N$, $N \log^2 N$, and the cluster

25

sizes used in both cases is $\log^2 N$. The plot in Figure 4 shows this comparison. The system using RWD performs much better irrespective of the loads assigned to it and the performance advantage increases as the load increases.

5.3 Performance Under Node Failures

An important aspect of our architecture is its performance under node failures. The key parameters of interest are: percentage of tasks that are successfully completed and are retrieved by the owner of the job, increase in completion time due to replication, and performance under varying failure rates. For these experiments, we use $K = \log^2 N$, $n = N \log^3 N$ and $N = 10,000$ nodes.

5.3.1 Node Failure Model

Node failures are modeled using a parameter α, which represents the fraction of nodes that fail in the time it would have taken for the job to complete in an environment without failure. This definition is useful for modeling node lifetime in comparison to the lifetime of the job. Node lifetimes are modeled as zipf random variables that are correlated to the degree of the node. The parameter α is used to normalize these lifetimes in relation to the job makespan time.

The number of nodes in the network is kept roughly constant by matching the arrival and failure rates. The size of the P2P does not have any implication on the load balance achieved by our randomized algorithms. However, the size of the P2P system does impact two important parameters, which are the length of the random walk and the size of the RS-set. In our system, these parameters are estimated based on the largest size of the network and hence remain appropriate if the size of the network shrinks. This is because, for uniform sampling, the only requirement is on the minimum length of the walk; a walk length longer than the minimum still yields a uniform sample (cf. Section 4.1.2). Similarly, if the RS-sets have a larger size than required, the intersection between two RS-sets is still guaranteed with high probability. Certainly, longer walks and larger sized RS-sets are not optimal with respect to minimizing system overhead, however, correctness takes precedence. Our model for node failures, which keeps network size to be roughly constant, is focused to cleanly exhibit the relative performance evaluation of the two replication schemes.

5.3.2 Evaluation

We compare the performance of the two schemes described in the paper, namely replication-at-initiation and multi-step replication. For the replication-at-initiation method, we use a replication factor of two (i.e., two copies of the replicated processes are submitted to the system). The effect of higher replication levels is dis-

cussed later. We let the owner query the system (through the rendezvous service) for its running tasks often enough so that it knows almost instantaneously if no more of its tasks are running. This is unrealistic in a real-world environment, but our results here are meant to provide a bound on how well the system can perform. For the multi-step replication method, the system assumes an efficiency of 40% and estimates the job makespan as was described in the protocol. It performs the multi-step replication until at least one copy of each task's result is obtained, and then stops. The parameter α is varied from 0.05 to 0.5.
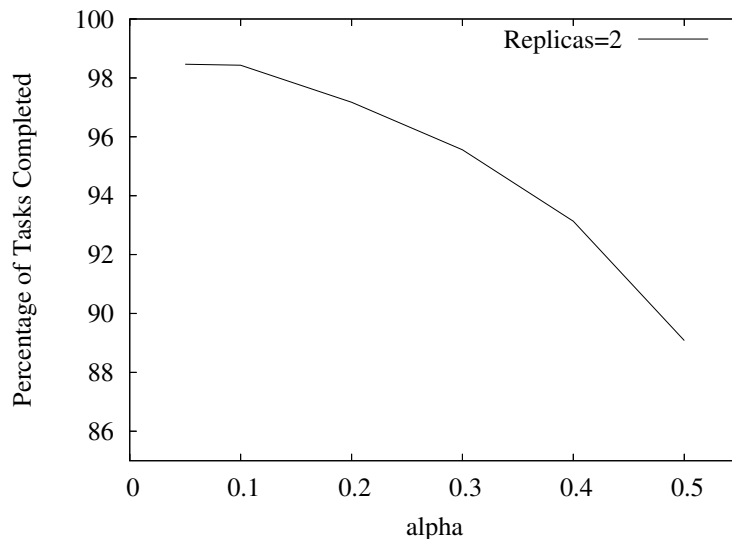

Fig. 5. Percentage of unique task results successfully retrieved by the owner for the replication-at-initiation scheme.

The plot in Figure 5 shows the percentage of unique tasks successfully retrieved by the owner for the replication-at-initiation scheme. As the fraction of failing nodes increases, the success ratio decreases rapidly. In comparison, the multi-step protocol achieves a 100% completion rate. The plot in Figure 6(a) compares the job completion time of the replication-at-initiation with two replicas and the multi-step algorithm. The job completion times are normalized with respect to the time it would take the job to finish in an environment with no failures. The plot shows that for $\alpha \leq 0.2$ the multi-step algorithm performs better. However, as α increases, the time taken by the multi-step algorithm increases. On the other hand, the time for the replication-at-initiation scheme decreases gradually. This gradual decrease is because many tasks are lost with the failing nodes and thus the time for the network to have no tasks executing is reached earlier. Note also that the time taken by the multi-step technique starts to level off when the rate of failure is higher. This is because the multi-step technique has an increasing degree of replication at each step. Once the replication level becomes high, it compensates for the high failure rate.

The plot in Figure 6(b) shows the average number of replicas found. We observe that the average number of replicas found by the replication-at-initiation scheme de-
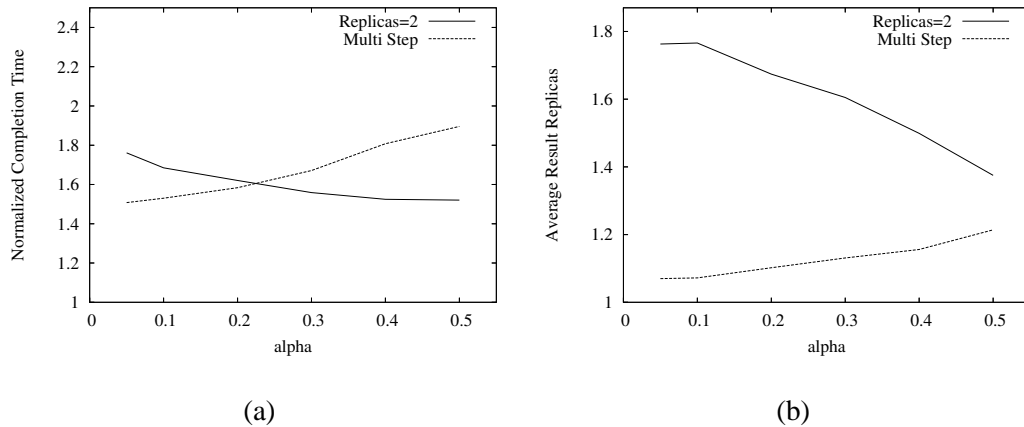
Fig. 6. (a) Completion time for the two schemes. (b) Number of replicas found by the master for the two schemes.

crease because of increased failures. On the other hand, for the multi-step scheme, each iteration of replication submission increases the replication rate. With high failure rates, the number of replicas submitted is higher and thus the average number of replicas received is higher. Quantitatively, the multi-step replication scheme has fewer redundant replicas and hence uses resources more efficiently. However, as discussed earlier the replicas received can be useful for validating results. Nevertheless, the multi-step scheme can be easily modified to increase the redundancy in a controlled fashion, simply by submitting replicated tasks even after one result for that task has been successfully received.
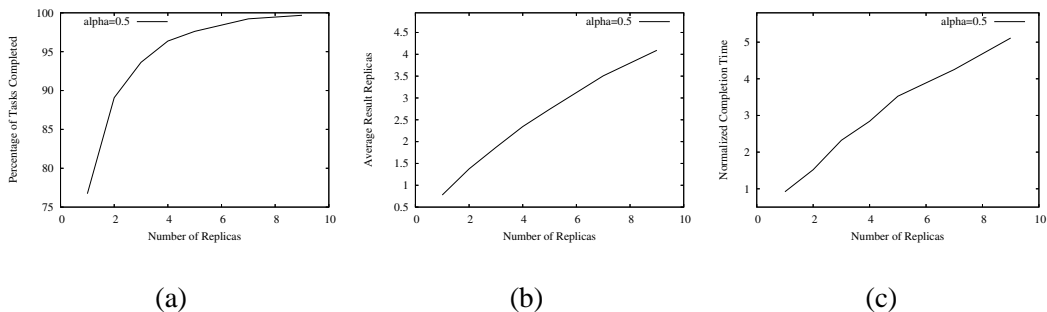

Fig. 7. Effect of varying replication level for failure rate $\alpha = 0.5$: (a) percentage of the jobs completed, (b) average number of replicas found by the master, (c) completion time normalized w.r.t. time required for job execution without failures.

To evaluate the benefit of using different levels of replication in the replication-at-initiation scheme, we repeat the experiment with $\alpha = 0.5$, while varying the replication level. The number of replicas submitted is increased from 1 (i.e., no redundancy) to 9 i.e., $\log N$. The plots in Figure 7 summarize the results. The plot in Figure 7(a) shows that increasing the replication level results in the number of tasks completed to asymptotically approach 100%. However, a perfect result is not achieved even with very high replication levels. The plot in Figure 7(b) shows that

average number of results retrieved from replicas is almost 50% of the number of replicas submitted. This is consistent with the failure fraction $\alpha = 0.5$. When more replicas are submitted, there is a higher chance of the replicas reaching a long lived node. Due to the zipf distribution such long lived nodes might have a much higher lifetime compared to the average. Finally, the plot in Figure 7(c) shows that the time for completion (i.e., no tasks of this job remain on the network) proportionally increases as the number of replicas increase. An important contrast can be drawn here to the performance of the multi-step technique. For $\alpha = 0.5$, the time taken by 7 fold replication, which achieves 99.2% completion, is almost 2.2 times more than the multi-step technique, which always achieves 100% replication.

6 Conclusion

In this paper, we present a distributed architecture for sharing processor cycles in unstructured P2P networks. The use of unstructured P2P networks is motivated by the success of massive real-world networks for file sharing. We present randomized algorithms for allocating tasks in the network, which achieve a good load balance and low job makespan. We analytically show that random job allocation using uniform sampling achieves good load balance. We present two protocols that incorporate redundancy for resilience against frequent node departures and validation of the execution output. The parameters that affect job throughput are discussed in the context of our allocation scheme. Our architecture includes a rendezvous service that allows job progress monitoring, aggregation of tasks, node reputation management, and context-based communication between oblivious hosts. We show that the rendezvous service provides probabilistic guarantees for locating resources.

Our algorithms are built on the premise of uniform sampling in an unstructured network. We show that random walks are ideal for random sampling, however, the resulting samples are affected by the topology of the network. We present an algorithm that allows uniform sampling via random walks irrespective of the underlying network topology. This is done by building a transition matrix for the walk in a distributed fashion. The resulting transition probability matrix also reduces the length of the random walk required to converge to uniform stationarity. The efficiency of the resulting cycle sharing system is evaluated using comprehensive simulation. The system is also evaluated with varying rates of node failures. The simulation results reflect the efficiency and robustness of our randomization based protocols.

References

[1] SETI@home., `http://setiathome.ssl.berkeley.edu/`.

[2] Genome@home., `http://www.stanford.edu/group/pandegroup/genome/`.

[3] distributed.net., `http://www.distributed.net/`.

[4] Gnutella, `http://www.gnutella.com`.

[5] Berkeley Open Infrastructure for Network Computing (BOINC)., `http://boinc.berkeley.edu/`.

[6] L. Lovasz, Random walks on graphs: A survey, Combinatorics, Paul Erdos is Eighty (Volume 2) (1996) 353–398.

[7] XtremWeb., `http://www.lri.fr/~fedak/XtremWeb/`.

[8] Condor, `http://www.cs.wisc.edu/condor/`.

[9] Globus, `http://www.globus.org/`.

[10] I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, Intl. J. Supercomputer Applications 11 (2) (1997) 115–128.

[11] Legion, `http://www.cs.virginia.edu/~legion/`.

[12] Avaki, `http://www.avaki.com/`.

[13] N. Kapadia, J. Fortes, PUNCH: an architecture for web-enabled wide-area network-computing, Cluster Computing: The Journal of Networks, Software Tools and Applications; special issue on High Performance Distributed Computing. 2 (2) (1999) 153–164.

[14] Sun Microsystems, `http://wwws.sun.com/software/gridware/`.

[15] A. Butt, X. Fang, Y. Hu, S. Midkiff, Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing, in: Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04), San Jose, CA, 2004.

[16] M. Castro, P. Druschel, Y. Hu, A. Rowstron, Proximity Neighbor Selection in Tree-Based Structured Peer-to-Peer Overlays, Tech. Rep. MSR-TR-2003-52, Microsoft Reseacrh (2003).

[17] A. Rowstron, P. Druschel, Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems, in: Proceedings of the 2001 ACM SIGCOMM, San Diego, CA, 2001, pp. 247–254.

[18] R. Gupta, A. Somani, CompuP2P: An Architecture for Sharing of Computing Resources in Peer-to-Peer Networks With Selfish Nodes, in: Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems, Cambridge, MA, 2004.

[19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, in: Proceedings of the 2001 ACM SIGCOMM, San Diego, CA, 2001, pp. 149–160.

[20] R. Mason, W. Kelly, G2-P2P: A Fully Decentralised Fault-Tolerant Cycle Stealing Framework, in: Proceedings of the Australasian Workshop on Grid Computing and e-Research, Newcastle, Australia, 2005.

[21] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing, Tech. Rep. UCB/CSD-0101141, UC Berkeley, Computer Science Division (April 2001).

[22] C. Kruskal, A. Weiss, Allocating Independent Subtasks on Parallel Processors, IEEE Transcations on Softaware Engineering 11 (10) (1985) 1001–1016.

[23] D. Malkhi, M. Reiter, R. Wright, Probabilistic Quorum Systems, in: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC'97), Santa Barbara, CA, 1997, pp. 267–273.

[24] P2P Trust and Reputation Papers, http://www.cs.umd.edu/~slee/p2p-trust-paper-list.html.

[25] BOINC Security, http://boinc.berkeley.edu/security.php.

[26] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.

[27] P. Brémaud, Markov Chains Gibs Fields, Monte Carlo Simulation, and Queues, Springer-Verlag, 1999.

[28] S. Saroiu, K. P. Gummadi, S. D. Gribble, Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts, Multimedia Systems 9 (2) (2003) 170 – 184.

[29] A. Awan, R. Ferreira, S. Jagannathan, A. Grama, Distributed Uniform Sampling in Unstructured Peer-to-Peer Networks, in: Proceedings of HICSS-39, Kauai, HI, 2006.

[30] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, Equations of State Calculations by Fast Computing Machines, J. Chem. Phys. 21 (1953) 1087–1901.

[31] W. Hastings, Monte carlo sampling methods using Markov chains and their applications, Biometrika 57 (1970) 97–109.

[32] Limewire., http://www.limewire.com/english/content/glossary.shtml.

[33] S. Saroiu, P. K. Gummadi, S. D. Gribble, A Measurement Study of Peer-to-Peer File Sharing Systems, in: Proceedings of Multimedia Computing and Networking 2002 (MMCN '02), San Jose, CA, 2002.

[34] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, Search and replication in unstructured peer-to-peer networks, in: Proceedings of ACM ICS'02 Conference, New York, NY, 2002.

[35] PlanetLab, http://www.planet-lab.org/.