USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

Java, Peer-to-Peer, and Accountability: Building Blocks for Distributed Cycle Sharing

Ali Raza Butt, Xing Fang, Y. Charlie Hu, and Samuel Midkiff
Purdue University
West Lafayette, IN 47907
{butta, xfang, ychu, smidkiff}@purdue.edu

Abstract

The increased popularity of grid systems and cycle sharing across organizations leads to the need for scalable systems that provide facilities to locate resources, to be fair in the use of those resources, and to allow untrusted applications to be safely executed using those resources. This paper describes a prototype of such a system, where a peer-to-peer (p2p) network is used to locate and allocate resources; a Java Virtual Machine is used to allow applications to be safely hosted, and for their progress to be monitored by the submitter; and a novel distributed credit system supports accountability among providers and consumers of resources to use the system fairly. We provide experimental data showing that cheaters are quickly identified and purged from the system, and that the overhead of monitoring jobs is effectively zero.

1 Introduction

This paper describes a prototype of a complete system that allows the sharing of cycles across network connected machines. For any such system to be successful, certain core functionality must be provided to both applications and hosts: (i) the ability for an application to discover a machine (or cluster of machines) capable of hosting it (resource management and discovery); (ii) the ability for an application to run on a wide variety of machines without change (portability); (iii) the ability of a host machine to accept an application from an untrusted source, and execute it without being damaged (safety); and (iv) a mechanism for maintaining information about resources provided and consumed, and for ensuring fairness in the use of resources (accountability). These four functions must be accomplished in a distributed, scalable manner to enable large networks of machines and resources.

Resources that are available but that are not easily discovered are useless. *Peer-to-peer* (p2p) networks have achieved widespread use as a content discovery mechanism. We propose using these same mechanisms for resource discovery and job assignment for our cycle sharing framework. Moreover, because p2p networks are self-organizing, it is easy for nodes to join, and leave, without the necessity of a central administrative organization and human intervention.

The use of Java is extremely convenient, if not essential, for the second and third of these functions (portability and safety) and it makes the accounting significantly easier. Because overspecifying a host machine's characteristics reduces the number of viable execution targets for an application, the portability across execution environments provided by Java is essential. Moreover, Java's built-in sandboxing technology, and rich security infrastructure, allow applications of varying degrees of trust to be hosted without each host providing additional security mechanisms. Both of these attributes significantly lower the cost and risk for producers and consumers of cycles to join a network of shared resources. And research [25, 26] shows that there are no inherent reasons for not using Java for high performance computing.

Finally, a community of pooled resources will survive only as long as members are treated with a high (but not necessarily perfect) degree of fairness. In the physical world, money is used as a conveyor of information about one's contribution to the economy. Credit reports allow providers of services to judge the likelihood that they will be paid for those services and to hold consumers accountable for their debts. Incremental payment schemes are used in many large activities to bound the amount of risk for both providers and consumers of resources to the size of the incremental payment. In this paper we outline a low-overhead Java based mechanism to allow users to monitor the progress of their applications, and to determine if they are comfortable making partial payments for the progress of a job. Our credit mechanism provides a distributed, scalable system for making and accepting these (possibly partial and incremental) payments (credits, in the language of this paper.) Moreover, our system supports two other important functions: we allow both users and resource providers to determine, using their own criteria, the credit worthiness of others; and we allow credits held by one entity (user) to be traded to another, and used by that other entity to acquire resources or reduce its volume of outstanding credit. Just as the larger economy can function well with a certain amount of fraud and noise in transactions and accounting, so should economies involved in sharing of computational resources. Thus our goal is not to produce perfectly secure system, but instead sufficiently good systems to enable wide scale sharing of computational resources.

This paper makes the following contributions:

- A novel method for monitoring the progress of a Java application with low overhead that leverages important features of Java Virtual Machines (JVM);
- A novel method for issuing credits that is scalable and checkable by all participating nodes in a system;
- A system that allows the sharing of computational resources that builds on the Java properties of portability and safety, the credit system described in this paper, and scalable p2p networks;
- Experimental data showing the practicality of these techniques.

The rest of paper is organized as follows. Section 2 presents our proposed scheme for enforcing fairness in p2p cycle sharing systems. Section 3 discusses the details of our prototype implementation, and Section 4 measures the overhead and the effectiveness of our proposed scheme. Finally, Section 5 presents some related work, and Section 6 provides concluding remarks.

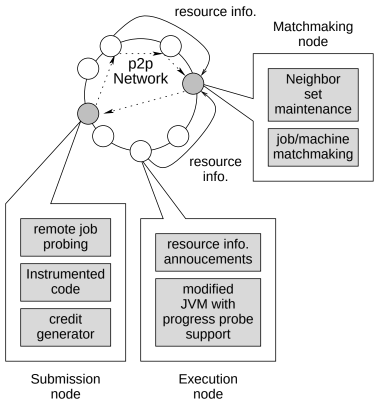


Figure 1: Overall design of the proposed scheme. Each node can be a submission node, or an execution node for submitted jobs.

2 System Design

This section first gives an overview of the system design, and then describes in detail how each of the building blocks implements the properties necessary for a usable cycle sharing system.

2.1 Overview

In order to obtain a cycle sharing system that prevents disproportionate consumption of resources, the system has to simultaneously provide the consumer with assurances that jobs are making progress on remote machines, and the resource provider with assurances that resource usage will be compensated. Figure 1 shows the design of our system. It uses a p2p substrate for two purposes: to organize all the participating nodes, and to locate remote nodes with available compute cycles. In the following we assume a remote node with free cycles has already been selected for remote execution.

Every participating node can be a resource consumer (submission node) or a provider (execution node). A node can fill both roles simultaneously as well, e.g., its jobs can be running on some remote nodes while jobs from other remote nodes are running on it. As a consumer, a node implements the following functions: (i) it implements a probing system that allows it to query a *reporting module* (a job that listens for, and accumulates information sent by the beacons which monitors the progress of an application) for progress reports on remote jobs; (ii) it has access to a special compiler that processes the information sent by beacons and check it for validity. In our current implementation all information sent by beacons is considered valid, and no information is extracted, but our design allows for more sophisticated reports (see Section 2.4.1 for details); (iii) it implements an accounting system to record the fact that it has consumed cycles on other nodes, and to issue digitally signed credits to the hosting machine when necessary. These reports are stored in the p2p network, and can be viewed by all nodes in the network.

As a resource provider, the execution node supports a trusted JVM whose dynamic compiler inserts *beacons* that send information about the progress of the application to the reporting module, mentioned above. The reporting module issues progress reports using this information.

Remote cycle sharing works as follows. First, the submission node creates a job to be run on remote resources. The p2p network is queried for a possible host node, the credit information for the node is checked, and if acceptable the job is submitted to the node. The VM and its dynamic compiler on the host node insert instrumentation and beacons into the program, which begins execution. The submitting node then periodically queries the reporting module (which can run on any node that can communicate with the job on the host node and with the submitting node). If the submitting node finds the job to be making progress, it issues a credit to the execution node. Credits are not issued if the job is not making satisfactory progress. If the submission node tries to cheat and not issue a credit to the host node, the host node can evict the job. Therefore, self interest motivates the submitting node to issue credits, and the host node to run the program.

2.2 Scalable resource management through p2p networks

To enable fault-tolerant, load-balanced sharing of compute cycles among the participating nodes, we use a structured p2p overlay network to organize the participating nodes and locate available compute cycles on remote nodes for remote execution.

2.2.1 Distributed Hash Tables

We briefly review current p2p overlay networks which previously have been used for supporting data-centric applications. Structured p2p overlay networks such as CAN[30], Chord[35], Pastry[32], and Tapestry[40] effectively implement scalable and fault-tolerant *distributed hash tables* (DHTs). Each node in the network has a unique nodeId and each data item stored in the network has a unique key. The nodeIds and keys live in the same namespace, and each key is mapped to a unique node in the network. Thus DHTs allow data to be inserted without knowing where it will be stored and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored. In the following, we give a brief description of one concrete structured overlay network, Pastry, on which our prototype is built. Detailed information can be found in [32, 7].

Pastry provides efficient and fault-tolerant content-addressable routing in a self-organizing overlay network. Each node in the Pastry network has a unique nodeId. When presented with a message with a key, Pastry nodes efficiently route the message to the node whose nodeId is numerically closest to the key, among all currently live Pastry nodes. Both nodeIds and keys are chosen from a large Id space with random uniform probability. The assignment of nodeId can be application specific; typically by hashing an application specified value using SHA-1 [14]. The message keys are also application specific. For example, when inserting the credit of a node into the DHT (Section 2.4) the message key could be some identifier that uniquely identifies that node.

The Pastry overlay network is self-organizing, and each node maintains only a small routing table of $O(\log N)$ entries, where N is the number of nodes in the overlay. Each entry maps a nodeId to an IP address. Specifically, a Pastry node's routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $(2^b - 1)$ entries each. Each of the $(2^b - 1)$ entries at row n of the routing table refers to a node whose nodeId shares the first n digits with the present node's nodeId, but whose $(n+1)$th digit has one of the $(2^b - 1)$ possible values other than the $(n+1)$th digit in the present node's nodeId. Each node also maintains a *leafset*, which keeps track of its l immediate neighbors in the nodeId space. The leafset can be used to deal with new node arrivals, node failures, and node recoveries as explained below. Messages are routed by

Pastry to the destination node in $O(\log N)$ hops in the overlay network.

2.2.2 Discovering free cycles

While p2p overlay networks have been mainly used for data-centric applications, our system exploits p2p overlays for compute cycle sharing. Specifically, it organizes all the participating nodes into an overlay network, and uses the overlay to discover available compute cycles on remote nodes.

To discover nearby nodes that have free cycles, our system exploits the *locality-awareness* property of Pastry [7] to maintain and locate nearby available resources to dispatch jobs for remote execution. This property means that each entry in a Pastry node n's routing tables contains the node n_c that is close to n among all the nodes in the system whose nodeIds share the appropriate prefix that would place them in that entry of n.

Periodically, each node propagates its resource availability and characteristics to its neighbors in the proximity space. This is achieved by propagating the resource information to the nodes n' in each node n's Pastry routing table rows. Node n' also forwards the resource information according to a Time-to-Live (TTL) value associated with every message. The TTL is the maximum number of hops in the overlay between n and the nodes that receive its resource information. Hence, the resource information is propagated to neighboring nodes within TTL hops in the overlay. Since Pastry routing tables contain only nearby nodes, this "controlled flooding" will cause resource information to be spread among nearby nodes in the proximity space. Each node that receives such an announcement caches the information in the announcement for its local matchmaking between jobs and available resources.

To locate a remote node for job execution, a node queries nearby nodes with available resources as accumulated in its local knowledge, taking proximity and credit-worthiness into account. The actual remote execution of the program and subsequent I/O activities are performed with the remote node directly, and do not go through the overlay.

Our p2p-based cycle sharing system tolerates node/network failures as follows. When the node for remote execution fails, the submitting node discovers an alternative node for re-execution. The fault handling can be implemented in the runtime system and made transparent to the user program. This is an important advantage over the traditional client/server model, where failure of a server implies explicit reselection by the client. For reasons of failure resiliency (or malicious node detection) via comparison of results from multiple nodes, a node may create multiple identical computations for remote execution.

2.3 Safety and portability through Java VMs

One of the goals of this project is to allow cycle sharing with a minimum of human-based administrative overhead. This requires that jobs be accepted from users who have not been vouched for by some accrediting organization. This, in turn, requires that submitted applications not be able to damage the hosting machine. Java's sandboxing abilities fit in well with this model. If submitters have undergone some additional verification (i.e. they are a trusted user), digital signature based security mechanisms can be used to allow potentially more harmful code to be executed, for example code that accesses the host's file system.

Java portability across different software and hardware environments significantly lowers the barriers to machines joining the pool of users and resources available on the network. Java's portability is in part because byte-code is well defined. Another reason for its portability is that much of the functionality that is provided by system libraries, and is not part of languages like C++ and Fortran (e.g. thread libraries and sockets), is provided by well specified standard Java libraries. As a consequence, in practice Java's interfaces to system services, e.g., sockets, appear to be more portable than with C++ implementations.

These attributes allow submitting nodes to have a larger number of potential hosts to choose from, and increases the probability that a program will execute correctly on a remote host.

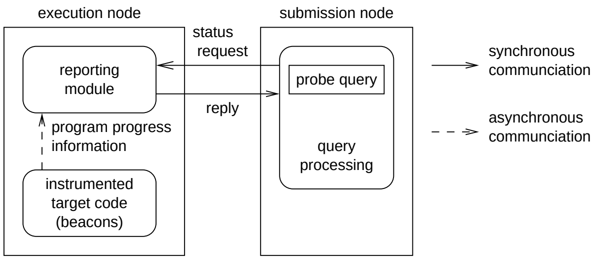

execution node
submission node
status request
reporting module
probe query
reply
program progress information
query processing
instrumented target code (beacons)
synchronous communciation
asynchronous communciation

Figure 2: Compiler instrumentation for monitoring job progress. The communication between the instrumented beacons and the reporting module is asynchronous, whereas it is synchronous between the querying node and reporting module.

2.4 Accountability through Java, p2p networks, and credit-worthiness

Accountability in our system is achieved through a system of credits. The storage and retrieval of these credits is accomplished through the p2p network, and the monitoring of long-running jobs to determine if credits should be issued is done using a JVM.

2.4.1 Compiler support for progress monitoring

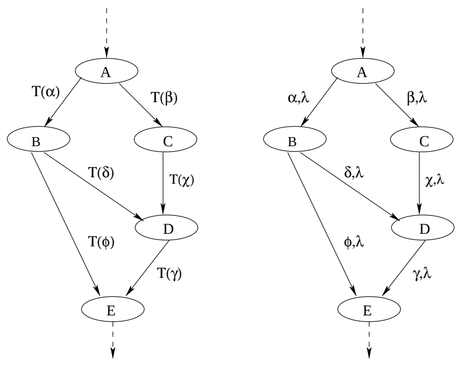
The basic idea behind the compiler assisted monitoring of the program execution is that the compiler will instrument the program with beacons which will periodically emit some indication of the program's progress. Because Java is a dynamically compiled language, commercial and research JVMs typically monitor the "hotness" of executing methods by either inserting sampling code that is periodically executed or executed on method entry, or by observing currently active methods. These techniques develop an approximation of the time spent in a method. Our system exploits this information to monitor the progress of the program and sends this information asynchronously to a separate program called the reporting module. The reporting module then buffers this information so that it can reply to queries from the program owner.

When a program owner queries the reporting module, the reporting module creates a progress report from the data it has so far collected, and replies to the program owner immediately. The owner can then process this report and determine whether its program is making progress. Figure 2 shows this setup. The advantage of having the reporting module is two-fold: (i) the application program does not have to suspend itself while waiting to be probed by the job owner, instead the data is buffered in the reporting module, and (ii) the design of the beacons is decoupled from the design of the queries. Moreover, the reporting module can run on the execution node, the submission node, or any other node that can communicate with both the execution and the submission node. For this reason, the reporting module is implemented as a separate process.

On a truly malicious, as opposed to overloaded or otherwise defective system, this simple beacon can be spoofed by the malicious system replaying old beacons. In the worst case, determining that a program on a remote system has run to completion and without tampering is as hard as actually running the program. It is our assumption in this project that we are not executing on truly malicious machines, rather we are running on machines that may be overcommitted, or that may be "fraudulently" selling cycles that don't exist in order to gain credits to purchase real cycles. Our goal is to uncover fraud and overcommitted nodes before the system is exploited "too heavily" by fraudulent or over-extended machines, not to prevent all fraud. Thus our system does not need to detect all fraudulent or overcommitted systems, but rather must allow fraudulent and overcommitted systems to be detected "soon enough". This is analogous to the goal of credit rating services in "real world" commerce, which is not to prevent any extension of credit to unworthy recipients, but rather to bound the extent to which they can receive credit to an amount that can be absorbed by the system. We stress that system is general enough to support either decentralized or centralized credit reporting mechanisms, depending on any legal requirements or requirements of the member nodes.

We are developing audit methods for better, albeit still not perfect, credit reporting. Figure 3(a) show a graphical representation of a program – it can be either a control flow graph or a calling graph. This graph can be treated as a transducer[1], or as a finite state automata (FSA) that accepts the language defined by the strings emitted by the transducer. In this model, the execution of the program corresponds to the transducer, with the reporting module implementing the recognizing FSA. The compiler will insert beacons to implement the

[1] A transducer is a finite state automata that emits information on state transitions.



(a) A flow or calling graph as a transducer

(b) The FSA corresponding to this transducer

Figure 3: A flow graph as a transducer and recognizing Finite State Machine.

transducer within the executing application and will also create the associated FSA.

Output emitted by the transducer can be simple, such as method names, or more complicated, such as method names, ranges of induction variables and the values of the induction variables themselves. However, this system may still not be secure since either another program understanding tool or a human could reverse-engineer the transducer in the application program and use that information to spoof the reporting module.

2.4.2 Issuing credits and assessing creditworthiness

To ensure the compensation of consumed cycles on node *B* by node *A*, we propose a distributed credit-based mechanism. There are two building blocks of our approach: (i) credit-reports which are entities that can be "traded" in exchange for resources, and (ii) a distributed feedback system which provides the resource contributors with the capability to check the credit history of a node, as well as to submit feedback about the behavior of a node. For simplicity, we assume that all jobs are equivalent in terms of the amount of resources they consume.

The distributed feedback database is built on top of the Distributed Hash Table (DHT) supported by the underlying structured p2p overlay. It maintains the feedback for each node regarding its behavior towards honoring credits. Any node in the system can access this information and decide whether to allow an exchange with a requesting node, or consider it a rogue node and avoid any dealings with it. In this way, a node can individually decide to punish a node whose consumption of shared resources has exceeded its contribution to other nodes by some threshold determined by the deciding node.

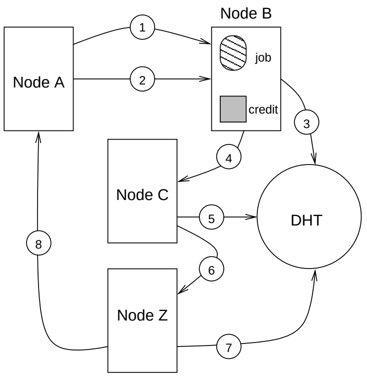


Figure 4: Various steps to ensure proper compensation of a contributed resource.

Figure 4 shows the various steps involved in ensuring that *B* is adequately compensated for its contribution. When A runs a job on B (1 in Figure 4), A will issue a (digitally signed) credit to B (2 in the Figure) (This credit is similar to the claim in Samsara [8]; it can be "traded" with other nodes for exchange of equivalent resources). A will give the credit a unique sequence number – unique in that A will issue no other credits with that number. B will digitally sign the credit and hash it into a repository in the p2p network (3 in the Figure). The credit hash is generated by hashing a function of A and the sequence number.

If B gives the credit to C (4 in the Figure), B will digitally sign the credit before giving it to C. C will digitally sign it, compute the hash based on (A, unique number) and store it (5 in the Figure). The storing node will replace the existing copy of the credit with the new copy. It knows it can do this since the end of the signing sequence is "B, B, C", i.e. the last-1 and last-2 signatures match, showing that the last-1 signer is the previous owner and is allowed to transfer the certificate.

If the certificate goes to Z (6 and 7 in the Figure) and returns to A (8 in the Figure), A destroys it. Since the end of the signing sequence is "Z, Z, A", the system knows that the transfer to A is valid and A is the owner, and therefore A can choose to have the credit destroyed. Note that this also allows credits to be destroyed by any owner (i.e. C above could have asked that the credit be destroyed, not saved) perhaps because of monetary payments, lawsuits, bankruptcy of the root signer, etc.

Because a credit hashes to a fixed location, attempts to forge credits (for example in a replay attack) will leave multiple copies of the credit (identified by its unique sequence number) in the same DHT location, and the second forged credit will not be saved. Thus if B tries to give the same credit d to both C and Z, one would be rejected (say Z's) and Z could then refuse to run B's job. Should a malicious node save both credits, a node checking on the credit worthiness of the issuing node can determine that there are two credits with the same number, and either ignore or factor this into its evaluation of both the issuing node *and* node B.

The feedback information is used to enforce contribution from selfish nodes as follows. In Figure 4, before node B executes a job on behalf of node A, it retrieves all the feedback for A from the DHT, verifies the signatures to ensure validity, and can decide to punish A by refusing its job if A's number of failures to honor credits has exceeded some threshold determined by B. We note that this system allows independent credit rating services to be developed that a submitting node can rely on for evaluating the credit-worthiness of a host.

2.4.3 Potential threats and vulnerabilities

It is our assumption in this project that we are not executing on truly malicious machines, rather we are running on machines that may be overcommitted, or that may be "fraudulently" selling cycles that don't exist in order to gain credits to purchase real cycles. Thus we limit out discussion of potential threats to be from such misbehaving nodes in the following.

The first scenario deals with the timing between the issuing of credits and the running of submitted jobs. In particular, a running node may refuse to spend further cycles upon receiving credits from job submitting nodes, and conversely, the submitting node may refuse to issue credits upon hearing the completion of its remote job execution. To provide mutual assurance, we propose the use of incremental credit issuing. Consider the case when A is submitting a job to run on B. Under the incremental credit issuing scheme, A gives incremental credits when it sees progress of its job on B. A might also choose to checkpoint its job when issuing an incremental credit to eliminate the chance of losing work that has been paid for. Secondly, A can also issue a negative feedback for B if it misbehaves. This is done as follows. A monitors its job on B at predefined intervals. On each interval that A finds its job stalled, it calculates a probability `q`, which increases exponentially with increasing number of consecutive failures of B to allow the job to progress. A then issues a negative feedback for B with the probability `q`. This scheme allows B to have transient failures, but punishes it for chronically cheating. Conversely, the case where A refuses to honor its issued credit is already addressed by the distributed feedback mechanism.

The next scenario results from the fact that if each feedback is inserted in the DHT only once, even though DHT replicates the credit among nodes nearby in the nodeId space, if the node storing the main replica acts maliciously, the credit information may not be retrieved correctly. To overcome this, we purpose inserting each credit into the DHT multiple times by making use of a predetermined sequence of salts known to all the participants. For each known salt, a credit hash is generated by hashing the concatenated function of the issuer's nodeId, the sequence number, and the salt. The signed credit is then inserted into the DHT. In this way, each credit will be available at multiple mutually-unaware nodes. When a participant intends to verify the integrity of another participant, it can choose to retrieve multiple copies of the credit instead of one. The number it attempts to retrieve can vary between one and the maximum copies that are allowed to be inserted into the system. Once these copies are retrieved, the node can compare them for trustworthiness. In case of mismatch, a majority vote can be adopted and the version of the credit that has the highest number of occurrence is assumed to be the trusted one. This potentially increases the amount of data that is inserted and retrieved from the DHT. However, the size of credit is usually small, and the additional benefits of having multiple insertions outweigh the increase in the message overhead.

Another problem may arise where the credits are never "traded" back to the issuing node. This does not indicate misbehavior, but can result in a large number of un-utilized credits accumulated in the system. Therefore, where possible, a node tries to "trade" a remote credit it holds, rather than issue its own credit. Moreover, each credit-report has a timestamp, and nodes can determine how old a

Module name	Functionality
`announceC`	Creates resource information announcements and sends them to the neighboring nodes.
`dbaseC`	Manages the local knowledge base on a node.
`matchmakerC`	Finds suitable nodes for requested job runs from available remote nodes.
`execC`	Sets up the execution environment for a job on a matched node and initiates the execution.

Table 1: Modules in the prototype implementation and their functions.

credit is. In case the age of a credit-report originated from A increases more than a system-wide threshold, the credit holding node will try to exchange it with A first, before trying to locate some other resources in the network. This will help in reducing the number of credit-reports in the system. Intra-organizational networks can clear the system periodically using internal budgeting procedures. We note that the load imposed on the system by large amounts of circulating credit is much less than in systems like Samsara, where credit takes the form of physical disk space held hostage, and consequently reduces the amount of system resources available for other purposes.

3 Implementation

We have built a prototype of our proposed scheme for p2p based resource discovery and accountability using Java 1.4.2 API specification. We utilize the Pastry [32] API for p2p functionality, and PAST [31] for storing the distributed feedback in a fault-tolerant and distributed manner. The implementation is done on nodes with Pentium IV 2 GHz processors, 512 MB RAM, running Linux kernel 2.4.18, connected via 100 Mb/s Ethernet. The prototype can be divided into various software modules as listed in Table 1.

An interesting observation in the implementation process was the duplicated reception of the same node's resource announcements at a node because a node may be on the routing tables of multiple remote nodes. To prevent duplicate messages from flooding the system, each node assigns a 32-bit sequence number to its announcements. The number is chosen at random at the start of a node, and increases monotonically for the life of the node. When an announcement is received, its sequence is first compared with the sequence number of the last message received from the originator. A sequence number with a value equal to or less than the last seen message implies that the message is a duplicate and can be discarded.

Our execution node uses an augmented version of the Jikes RVM [1], running on adaptive configuration. In the adaptive system, there exists an instrumentation framework that increments method invocation counters as the application proceeds. The counters are then stored in a database whose contents are examined by the controller thread to help make recompilation decisions. Since the application progress reports that we require can be inferred from this database, we do not have to add inline code into the application to determine it. Instead, we augment the system with a progress monitoring thread. In other words, the beacon is implemented as a thread that periodically looks into the database and sends the method invocation counter values to the reporting module, together with the timestamp. This information serves as an indicator for job progress. The interval between successive reports is a parameter that is specified when remote jobs are submitted. We will examine the effect of this interval on the execution overhead in our experimentation.

In our implementation the reporting module is located on the same host as the execution node. This enables the reporting module to record and report the system load information on the execution node as well, which provides further grounds for the probing module on the submission node to determine whether the remote host is cheating. (If the host is heavily loaded, we might assume the host is not cheating even though the progress is reasonably small). Also, hosting the reporting module on the same node incurs additional overhead, and we are able to study its effects through experimentation.

Communications between the execution node and the reporting module, and between the reporting module and the probing module, are implemented through UDP packets. We choose to use socket-based communication because it enables us to move the reporting module off the execution node if

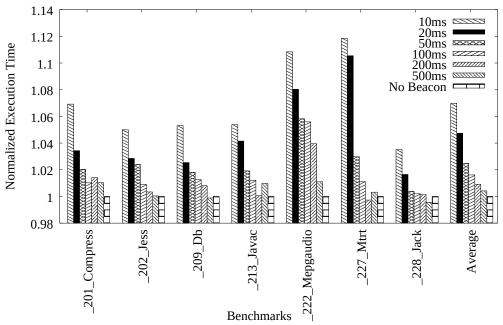


Figure 5: Overhead of beacons with a remote reporting module.

needed, and UDP is preferred because it has a lower overhead. As we are only interested in getting an indicator of application progress, losing some packets occasionally is not a big problem.

4 Evaluation

This section gives experimental results gathered by executing a modified version of the Jikes RVM on hardware, and by simulation studies of a p2p overlay network.

4.1 Instrumentation overhead

In this section, we determine the effect of progress monitoring on application performance by studying the overheads of the beacons and the reporting module.

To study the overhead of beacons alone, we first run the reporting module on a different node than the execution node. Figure 5 shows the overhead of adding the beacon thread in the virtual machine. Experiments were run with the SpecJvm98 benchmark suite on data sizes of 100. Instrumentation results were reported to the remote reporting module at time intervals of 10, 20, 50, 100, 200 and 500 milliseconds, respectively. We observe that on average, when the time interval between successive reports is 500ms, the performance impact is less than 1%. For an actual system, reporting intervals would be in minutes or seconds, so the overhead on the program performance would be effectively zero. Notice that this is the overhead of monitoring the program progress only. The cost of instrumentation in the Jikes RVM system, which we are leveraging for our technique, is about 6.3% [3], and can be lowered with larger instrumentation intervals.

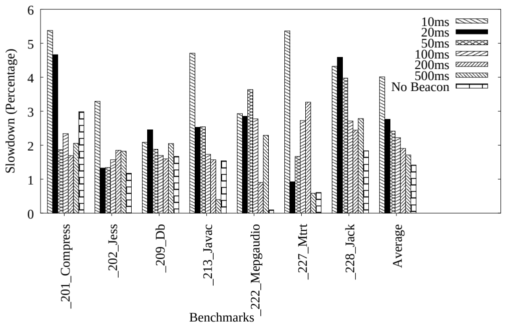


Figure 6: Overhead of hosting the reporting module on the execution node.

Next we determine the overhead of hosting the reporting module on the execution node. We compare the performance of this setup with one that uses a remote reporting module (as in the previous case). Figure 6 shows the results. There are three sources for the overhead: network traffic to and from the reporting module, cost of periodically collecting host system load information, and overhead of maintaining the application progress and system load database. Application execution on the host slows down as a result of competition for CPU cycles. The first and last kinds of the overhead will grow linearly with the number of remotely executing jobs and they also grow as instrumentation and probing frequencies increase. The cost of system load information is proportional to the interval of load probing. In our experiment we assume one running VM, one probing node, and a load test interval of 200ms. We get the overhead for the job running on the VM. We did not do tests with multiple VMs on the same execution node because in that case the overhead is difficult to express in terms of the slowdown, and multiple working VMs compete for CPU cycles among themselves. The presence of the reporting module, without beacons, causes a slowdown of 1.4%. Increased reporting density incurs additional slowdowns, but again with realistic reporting frequencies, we can assume that part of the overhead to be zero.

4.2 Simulation results

In this section, we evaluate the effectiveness of our p2p scheme on discovering appropriate resources for execution of jobs and on enforcing fairness in cycle

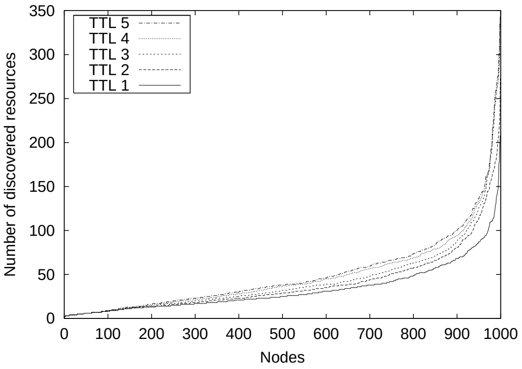


Figure 7: The number of resources available at each node with increasing TTL. The nodes are sorted in increasing order of discovered resources.

sharing by simulating a network of 1000 nodes. We developed a simulator of our proposed system on top of Pastry running with the direct communication driver, which allows the creation of multiple Pastry nodes on a single machine. We used the transit-stub Internet model [39] to generate an IP network with 10 stub domain routers, 100 transit domain routers, and attach 1000 nodes randomly to the 100 stub domain routers.

In the first set of simulations, we measure the effectiveness of our protocol for disseminating resource announcements. We assume each node has some resource to announce to the network, and uses the defined protocol to send the announcements. We run the simulation five times with TTL varying from 1 to 5. For each simulation, we measure the number of remote nodes whose announcements reach any given node. Figure 7 shows how the number of "discovered" nodes by each node increases with the TTL value. Increasing TTL results in more resources being discovered, but it also implies that resources may be far away in the network proximity space.

To measure the effectiveness of our credit-based scheme for enforcing fairness, we simulate cycle sharing among the 1000 nodes, with 500 nodes actively submitting jobs, while the rest of the nodes only contribute cycles. The TTL is fixed at three for this set of observations. Each of the 500 submitting nodes is fed with a randomly generated sequence of 100 jobs of equal length, each running for nine time units. The interarrival time between the jobs follows a uniform distribution between 1 and 17 time units. We compare the job throughput of three scenarios: (1) all 500 submitting nodes are honest; (2) 250 submitting nodes are honest and 250 submitting nodes cheat by only submitting jobs and never accepting remote jobs, and the fairness mechanism is not turned on; and (3) same as (2) but with the fairness mechanism turned on. For case (3), the threshold for detecting cheating nodes is set to three, i.e., a cheating node can run up to three jobs without compensating the system, but when it attempts to run more jobs, other nodes ignore its requests for resources. The job throughput under these scenarios are shown in Figures 8, 9, and 10, respectively.

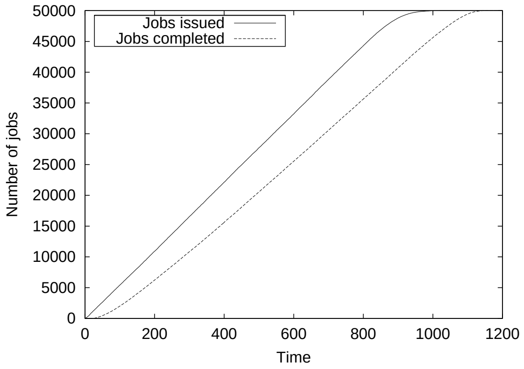


Figure 8: The number of jobs issued, and completed over the period of our trace. Every node contributes resources to the system and is fair.

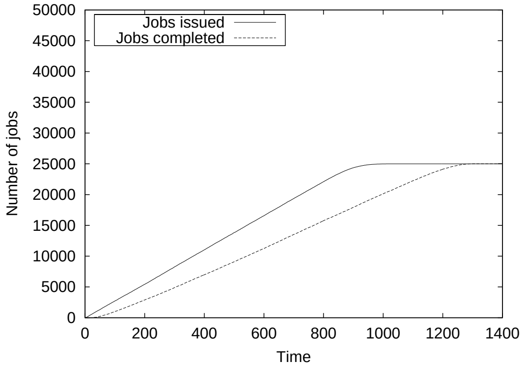


Figure 9: The number of jobs issued, and completed over the period of our trace for the non-cheating nodes. The feedback system is disabled.

Figure 8 shows the number of jobs issued and the number of jobs completed against our simulated time when all 500 nodes are honest. It is observed that the jobs do not have to wait if free resources are available. The number of completed jobs closely follow the number of issued jobs. The slight increase in the difference over time is due to the fact the more jobs were requested than the available resources. However, all jobs completed at about the 1150th time unit, only 150 time units after all the jobs were issued.

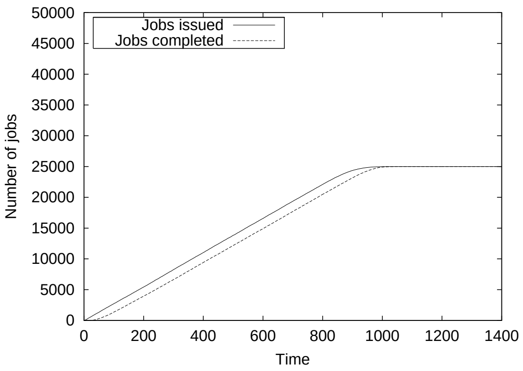


Figure 10: The number of jobs issued, and completed over the period of our trace for the non-cheating nodes. The feedback system is enabled.

Figure 9 shows when 250 submitting nodes cheat and the credit-based mechanism is not turned on, the non-cheating nodes experience a larger delay in their job completion. There is a delay of 345 time units for all the jobs to complete. This is significant considering that the job lengths are only 9 time units.

Finally, Figure 10 shows the credit-based mechanism effectively isolates the cheating nodes and the jobs from the non-cheating nodes made more progress compared to the case without the credit-based mechanism. Note that the job delay in this case is less than that in Figure 8, because here 250 non-cheating nodes were sharing cycles of a total of 750 nodes, where as in the latter case 500 nodes shared cycles of 1000 nodes. This simulation shows that the credit-based mechanism quickly prohibits cheating nodes from consuming other nodes' cycles.

5 Related work

We discuss related work on cycle sharing over the Internet, on compiler instrumentations, and on fairness in p2p storage sharing systems.

Cycle sharing over the Internet The idea of cycle sharing among a large number of administratively independent, geographically dispersed, off-the-shelf desktops is popularized by the SETI@home [34] project. Similar approaches for solving large scale scientific problems are also adopted in systems such as Distributed.NET [10], Entropia [12], Genome@home [18] , and Nile [28] . These systems implement a central manager that is responsible for the distribution of the problem set, and the collection and analysis of the results. Users typically download the client programs manually and then execute them on their resources. The client programs are specially developed applications that the resource owners have to explicitly trust [6]. The clients periodically contact the central managers to provide results and to receive further data for processing. The clients are pure volunteers in nature, i.e., they do not receive any resource contribution for their own tasks. The aim of our project is to provide all nodes in the system with the capability to utilize shared resources. This provides an incentive for more resource owners to contribute resources to the system, hence increasing the instantaneous compute capacity of the system.

Various grid platforms also share the same goal of distributed sharing resources. Condor [24] provides a mechanism for sharing resources in a single administrative domain by harnessing the idle-cycles on desktop machines. Globus [15] and Legion [20] allow users to share resources across administrative domains. However, the resource management is hierarchical, and the users have to obtain accounts on all the resources that they intend to use [6]. PUNCH [23] decouples the shared resource users from the underlying operating system users on each resource, hence eliminating the need for accounts on all the shared resources. Sun Grid Engine [37] is another system that harnesses the compute powers of distributed resources to solve large scale scientific problems. However, all of these systems rely on some forms of centralized resource management and therefore are susceptible to performance bottlenecks, single-point of failures, and unfairness issues that our system avoids by using p2p mechanisms.

Fair peer-to-peer storage sharing The idea of enforcing fairness has been extensively studied in peer-to-peer storage systems, motivated by the usage studies [11, 33] which show that many users consume resources but do not compensate by contributing. CFS [9] allows only a specified storage quota for use by other nodes without any consideration for the space contributed to the system by the consumer. PAST [31] employs a scheme where a trusted third party holds usage certificates that can be used in determining quotas for remote consumers. The quotas can be adjusted according to the contribution of a node. Samsara [8] enforces fairness in p2p storage without requiring trusted third parties, symmetric storage relationships, monetary payment, or certified identities. It utilizes an extensive claim management which leverages selfish behavior of each node to achieve an overall fair system. The fairness in our cycle sharing system was motivated by Samsara. However, fairness in cycle sharing is more complex than in data sharing as once a computation is completed, the execution node has no direct means of punishing a cheating consumer. SHARP [17] provides a mechanism for resource peering based on the exchange of *tickets* and *leases*, which can be traded among peering nodes for resource reservation and committed consumption. Credits in our system are similar to *tickets* in SHARP. However, our system uses the credit reports to enforce fairness of sharing, and not as a mean for advance resource reservations.

There have also been efforts to design a general framework for trading resources in p2p systems. *Data trading* [5] is proposed to allow a consumer and a resource provider exchange an equal amount of data, and cheaters can be punished by withholding the data. The approach requires symmetric relationships and do not apply well to p2p systems where there is very little symmetry in resource sharing relationships. The use of micropayments as incentives for fair sharing is proposed in [19]. Fileteller [22] suggests the use of such micropayments to account for resource consumption and contribution. In [38], a distributed accounting framework is described, where each node maintains a signed record of every data object it stores directly on itself or on other nodes on its behalf, and each node periodically audits random other nodes by comparing multiple copies of the same records. The system requires certified entities to prevent against malicious accusations, and the auditor has to work for other nodes, without any direct benefit. Our system implements a distributed accounting system as well, where a node verifies credit reports of a remote node only when it has to do an exchange with it, which is a direct benefit.

Compiler instrumentation and proof carrying code The concept of compiler generated instrumentation and monitoring of program execution within a JVM is not new. The Sun Hotspot [29] compiler, compilers for the IBM JDK [36], and compilers for the Jikes RVM [1] all use either compiler generated program instrumentation or an examination of the active routines on the stack to determine *hot* methods. This in turn is similar to profiling [2, 4, 13] for collecting information about where time is spent during a program's execution. These efforts are orthogonal to our work, and could complement our work by providing a mechanism for collecting information about total program run time. Our novelty is in bootstrapping off the already existing monitoring of program executions supported by JVMs as part of their optimization strategy to allow the progress of a program to be remotely tracked. Other projects (e.g. [21]) have used instrumentation to collect data for performance purposes, and allows on-the-fly instrumentation of statically compiled programs.

The GRAM component of the Globus project [16] also monitors program execution, and uses this information to change the resource requirements of the application to give better quality of service. Our work differs in our motivation – we are using the monitoring to determine if we are getting a resource as promised; and in our implementation – the monitoring measures program performance via automatic instrumentation by a JVM and not by external measures or by programmer inserted callouts from the program. This allows us to not require individual programs to be adapted to our system in order to use it.

The ultimate goal of our work is related to proof carrying code [27]. We differ from that work in that the ultimate goal here is to have a program certify to the submitter, rather than the host, facts about its execution, with these facts bound to the program form itself.

6 Conclusion

We have described the design of a system, and our implemented prototype, that exploits the safety, portability and internal profiling capabilities of Java and Java Virtual Machines. This system allows a decentralized p2p network to be used to advertise and allocate resources, and contains a credit system that allows decentralized sharing of resources and evaluation of credit-worthiness. Our experimental results show that our fairness mechanisms work well to punish cheating nodes, and our monitoring of program progress has effectively zero overhead. Because of its decentralized nature, leading to low costs for entry and exit from the network, our system is ideal for constructing ad-hoc intra-organizational networks of pooled resources, and for constructing pools of resources for small business and educational purposes.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was supported in part by NSF CAREER award grant ACI-0238379 and NSF grants CCR-0313026 and CCR-0313033.

References

[1] B. Alpern and *al. et.* The Jalapeo Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 1997.

[3] M. Arnold. *Online Profiling and Feedback-Directed Optimization of Java*. PhD thesis, Rutgers, The State University of New Jersey, October 2002.

[4] T. Ball and J. R. Laurus. Efficient path profiling. In *Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] B.F. Cooper and H. Garcia-Molina. Peer-to-peer resource trading in a reliable distributed system. In *Proc. First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.

[6] A. R. Butt, S. Adabala, N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Grid-computing portals and security issues. *Journal of Parallel and Distributed Computing: Special issue on Scalable Web Services and Architecture*, 63(10):1006–1014, October 2003.

[7] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002, 2002. ⟨ http://research.microsoft.com/~antr/PAST/ localtion.ps ⟩ (17 Oct 2003).

[8] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, October 2001.

[10] distributed.net. distributed.net projects (11 April 2003). ⟨ http://www.distributed.net/projects.php ⟩ (28 September 2003).

[11] E. Adar and B.A. Huberman. Free riding on Gnutella. *First Monday*, 5(10), October 2000.

[12] Entropia, Inc. Entropia: Pc grid computing (16 June 2003). ⟨ http://www.entropia.com/index.asp ⟩ (28 September 2003).

[13] J. Fenlason and R. Stallman. GNU gprof manual (Nov 7, 1998). ⟨ http://www.gnu.org/manual/ gprof-2.9.1/gprof.html ⟩ (Oct 14, 2003).

[14] FIPS 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington D.C., April 1995.

[15] I. Foster and C. Kesselmann. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, Jan. 1997.

[16] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Proc. 8th International Workshop on Quality of Service*, 2000.

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. 19th ACM Symposium on Operating Systems Principles*, October 2003.

[18] Genome@home. Genome at home(26 September 2003). ⟨ http://www.stanford.edu/group/pandegroup/ genome/index.html ⟩ (29 September 2003).

[19] P. Golle, K. Leyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proc. Third ACM Conference on Electroni Commerce*, Tampa, FL, 2001.

[20] A. S. Grimshaw and W. A. Wulf. Legion – A View from 50,000 feet. In *Proc. 5th IEEE International Symposium on High Performance Distributed Computing(HPDC'96)*, Syracuse, NY, 1996.

[21] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proc. IEEE PACT*, pages 201–, 1997.

[22] J. Ioannidis, A. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proc. Sixth Annual Conference on Financial Cryptography*, Bermuda, 2002.

[23] N. H. Kapadia and J. A. B. Fortes. PUNCH: An architecture for Web-enabled wide-area network-computing. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2(2):153–164, Sep. 1999.

[24] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988)*, pages 104–111, San Jose, CA, 1988.

[25] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The NINJA project: Making Java work for high performance computing. *Communications of the ACM*, 44(10):102–109, October 2001.

[26] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000. IBM Research Report RC 21166.

[27] G. Necula. Proof carrying code. 1997.

[28] Nile. Scalable solution for distributed processing of independant data (18 June 1999). ⟨ http://www.nile.cornell.edu/index.html ⟩ (29 September 2003).

[29] M. Paleczny, C. Click, and C. Vick. The Java HotSpot server compiler. In *Proc. 2001 USENIX Java Virtual Machine Symposium*, 2001.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 161–172, San Diego, CA, 2001.

[31] A. Rowstron and P. Druschel. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles*, October 2001.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[33] S. Saroiu, G. Krishna, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. SPIE Conference on Multimedia Computing and Networking*, San Jose, CA, 2002.

[34] SETI@home. Search for extraterrestrial intelligence at home (29 September 2003). ⟨ http://setiathome.ssl.berkeley.edu/index.html ⟩ (29 September 2003).

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01)*, pages 149–160, San Diego, CA, 2001.

[36] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 180–195, 2001.

[37] Sun(TM) Microsystems. Sun ONE Grid Engine Software (26 June 2003). ⟨ http://wwws.sun.com/software/gridware/sge.html ⟩ (29 September 2003).

[38] T-W.J. Ngan and D.S. Wallach and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. Second International Workshop on Peer-to-Peer Systems*, Berkeley, CA, 2003.

[39] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE INFOCOM*, March 1996.

[40] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.