J Grid Computing (2006) 4: 265–286
DOI 10.1007/s10723-006-9039-3

A Fair, Secure and Trustworthy Peer-to-Peer Based Cycle-Sharing System*

Shuo Yang · Ali R. Butt · Xing Fang · Y. Charlie Hu · Samuel P. Midkiff

Received: 1 September 2005 / Accepted: 28 February 2006 / Published online: 27 June 2006
© Springer Science+Business Media B.V. 2006

Abstract The increased popularity of Grid systems and cycle sharing across organizations requires scalable systems that provide facilities to locate resources, to be fair in the use of those resources, to allow resource providers to host untrusted applications safely, and to allow resource consumers to monitor the progress and correctness of jobs executing on remote machines. This paper presents such a framework that locates computational resources with a peer-to-peer network, assures fair resource usage with a distributed credit accounting system, provides resource contributors a safe environment, for example Java Virtual Machine (JVM), to host untrusted applications, and provides the resource consumers a monitoring system, GridCop, to track the progress and correctness of remotely executing jobs. We present the details of the credit accounting subsystem and the GridCop remote job monitoring subsystem. GridCop and the distributed credit accounting system together enable incremental payments so that the risk for both resource providers and resource consumers is bounded.

Key words Peer-to-peer · Cycle-sharing · Grid · Fairness · Incentive · Monitoring · Trustworthiness

*This work was supported by NSF CAREER award grant ACI-0238379 and NSF grants CCR-0313026 and CCR-0313033.

S. Yang · A. R. Butt · X. Fang · Y. C. Hu (✉) ·
S. P. Midkiff
School of Electrical and Computer Engineering,
Purdue University,
1285 EE Building,
West Lafayette, IN 47907, USA
e-mail: ychu@purdue.edu

1. Introduction

Computational workloads for many academic groups, small businesses and consumers are bursty. That is, they are characterized by long periods of little or no processing punctuated by periods of intense computation and insufficient computational resources. By aggregating large numbers of computers and users, the resource demands are ‘smoothed out’ across sub-groups even as demand remains bursty within sub-groups. Centrally managed software projects [11, 17, 28, 34] have sprung up to access idle machines to perform computations that would be economically infeasible to solve on committed hardware. Centrally managed systems like Condor [25] and LoadLeveler [23] have been developed to allow resources to be aggregated within permanent or *ad hoc* organizations.

Centralized administration of resources exists because it allows a trusted entity – system administrators – to verify and track the trustworthiness of users given access to the resources, *and* it allows users to deal with a known, trusted entity. Certification of the user of a machine is almost always contingent on being an employee of the machine owner, or being certified by another organization which the user belongs to, and which is, in turn, trusted by the machine owner. This certification requires legal contracts that carefully delineate risks and responsibilities, staff to maintain accounts, accountants to monitor funding streams and tax consequences, and generally increases the overhead and real cost of acquiring and using computational resources. This in turn restricts the domain of applications that can be run on shared resources.

The irony of this situation is that the resources to be shared are extremely perishable – Cycles, bandwidth and disk space not used in the past do not create additional resources to be consumed in the future. The major value of these resources to their owner is the knowledge that they are available when needed. Therefore, contributing cycles to others during idle periods imposes little cost on a resource owner, while achieving additional processing power during 'peak' periods significantly increases the return on an existing investment.

On the other hand, the major costs of sharing unneeded cycles are the legal and administrative overheads. Eliminating these overheads would dramatically increase the quantity, and decrease the cost, of available cycles. Both the decreased cost and the ease of accessing cycles would increase the range of applications that could exploit them. Academics and research laboratories would have access to a vast array of machines for running simulations, benchmarking programs, and running scientific applications; small businesses would have machines available for data-mining sales, accounting and forecasting; and consumers would have machines available to perform computationally intensive, but low-economic value activities such as games and digitally processing home movies. Elimination of these overheads would allow automatic intermediation between consumers and providers of resources, allowing shared resources to blend seamlessly with locally owned resources.

The promise of increasing available processing power during peak periods by trading cycles during idle periods, without significant additional cost, would motivate people to join a cycle-sharing system, but only one in which the following difficult technical challenges have been solved.

1. How can resource consumers discover computational resources capable of hosting their job?
2. How can an application be run on a variety of resource provider machines without change (portability)?
3. How can a resource provider machine safely execute an application from an untrusted source (safety)?
4. How can resource providers be compensated (and cheaters punished), to enforce the fairness of the system resource usage?
5. How does the resource consumer know its job is being faithfully executed? How does the resource consumer know its job is making progress?

Such a system will unleash the potential of the massive computational resources that are going unused.

In a cycle-sharing system, a *submitter* machine, i.e. a machine submitting a job request to the system, plays the role of a system resource consumer; a *host* machine, i.e. a machine accepting and executing a job, plays the role of a system resource provider. We use these terms to refer to components in our system design.

Peer-to-peer (p2p) networks (e.g., CAN [29], Chord [36], Pastry [32], and Tapestry [40]) have achieved widespread use as a content discovery mechanism. We propose using the same mechanisms for resource discovery and job assignment to solve the first challenge. Moreover, because of the self-organizing feature of p2p networks, it is easy for nodes to join, and leave, without the necessity of a central administrative organization and human intervention, which in turn obviates the need for a central organization and human intervention, and lowers administrative overhead.

The use of Java is extremely convenient, if not essential, for overcoming the second and third of these challenges (portability and safety). We utilize Java's universal virtual machine execution environment feature to enable applications to run on a wide variety of physical machines without any change (portability); We utilize a Java virtual machine's sandboxing and security feature to give resource providers a safe execution environment to host untrusted applications (host safety). Both of these attributes significantly lower the cost and risk for producers and consumers of cycles to join a network of shared resources. Moreover, research [26, 27] shows that there are no inherent technical reasons for not using Java for high performance computing.

To solve the fourth challenge, we have designed a credit accounting system. A community of pooled resources will survive only as long as members are treated with a high (but not necessarily perfect) degree of fairness. Moreover, 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 computational resources. Thus our goal is not to produce a perfectly fair system, but instead to produce a sufficiently good system to enable wide scale sharing of computational resources. An incremental payment scheme is used in many modern economic activities to bound the amount of risk for both providers and consumers of a transaction to with the size of the incremental payment. Our credit system, along with the GridCop system, enables the risk for both submitters and hosts in our cycle-sharing system to be bounded by the size of an incremental payment.

We have developed a remote job progress and correctness monitoring system, GridCop, to solve the last challenge. The GridCop system allows a computation on a remote, and potentially fraudulent, host system to be monitored for progress and execution correctness. Monitoring progress and correctness is especially important to the submitter during a long running application because it provides submitters confidence that their jobs will be executed by a remote, otherwise untrusted host. Moreover, along with the above credit system, GridCop enables the use of the incremental payment scheme.

This paper makes the following technical contributions:

- We design a framework for large scale cycle-sharing systems, in which constructive participant behavior is motivated by self-interest, and which solves the five significant challenges listed above.
- We design a distributed credit mechanism that along with the GridCop remote monitoring system (for accountability) enables the fair usage of system resources, and enables the incremental payment scheme.
- We design and implement a prototype of the GridCop system to track the progress and correctness of a program executing on remote and potentially fraudulent host machines. The experimental results of GridCop, when remotely monitoring a standard benchmark suite of computations over a wide area network, show that the overhead on the host is less than 4.5%, and the cost of monitoring a job is less than 0.4% of that of running the same job locally on the submitter side.

The rest of the paper is organized as follows. Section 2 first gives an overview of our cycle-sharing system and then presents the four constituent components one-by-one: A peer-to-peer overlay based resource discovery mechanism (see Section 2.2); a JVM based safe host platform (see Section 2.3); a distributed hash table (DHT) based credit accounting system; and the GridCop remote job monitoring system. The last two of these components are discussed briefly in Section 2, and in more detail in the next two sections. Section 3 describes the credit feedback system and incremental payment scheme in detail. Section 4 describes the design and implementation of the GridCop system in detail and presents experimental results showing its low overhead. Finally, Section 5 discusses the related work and Section 6 concludes the paper.

2. Design of a Large Scale Self-Interest Motivated Cycle Sharing System

In this section, we first give an overview of our cycle-sharing framework in Section 2.1. In the

remainder of the section we present the four components and their functionality within our cycle-sharing system.

2.1. Overview of the Cycle-Sharing System

Our remote cycle sharing system works as follows. Every participating node can be a resource consumer (submitter node) or a provider (host node). A node can fill both roles simultaneously as well, e.g., its jobs can be running on remote nodes while jobs from other nodes are running on it. Each participating node installs a Java virtual machine (JVM) as the safe execution environment when it hosts a job. Each participating node installs the GridCop monitoring runtime library. Figure 1 shows the design of our system.

Before a submitter submits a job to the system, it passes the program through our code transformation tool which transforms it into a program that executes on the host machine (*H-code*) and a program that executes on the submitter machine (*S-code*). Then, the p2p network is queried for a possible *host* node, the credit information for this node in the accounting system is checked, and if acceptable, the job is submitted to the node for execution. Credit reports are stored in the p2p network, and can be viewed by all nodes in the network.

The H-code is the original program augmented with *beacons* and auxiliary code that send beacon information about the program to the submitter machine during execution. The S-code, executing on the submitter machine (or a trusted machine accessible from the submitter machine), uses this information to track the progress, and verify the correctness of, the program during its execution. The submitter incrementally issues digitally signed credits for the hosting machine to the accounting system to record the fact that it has consumed cycles. If the submitting node finds the job is 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 submitter node to issue credits, and the host node to run the program. Our remote program progress monitoring system, along with the accounting system, simultaneously provides the consumer with assurances that jobs are making progress on remote machines, and the resource provider with assurances that resource usage will be compensated. With the incremental payment scheme enabled by the GridCop system,

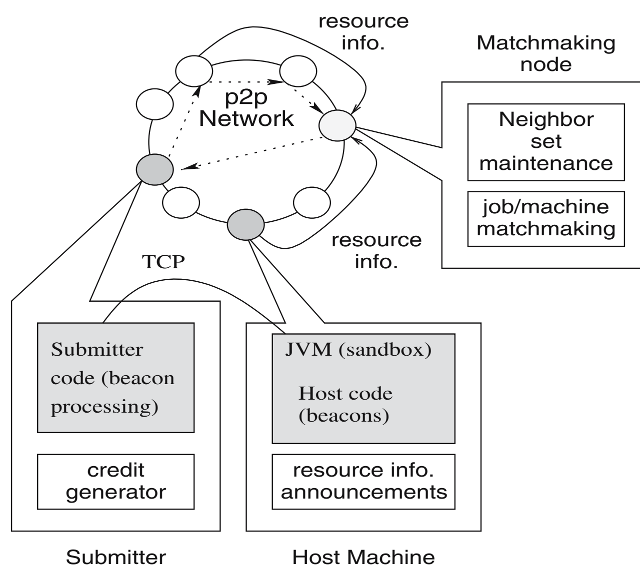


Figure 1 Overall design of the proposed scheme. Each node can be a submission node, or an execution node, for the submitted jobs.

the risk for both submitters and hosts is bounded to the size of an incremental payment.

2.2. Resource Discovery through Peer-to-Peer Networks

While p2p overlay networks have been used mainly for data-centric applications, our system exploits p2p overlays for compute cycle sharing. Specifically, it organizes all the participating nodes into a peer-to-peer overlay network, and uses the overlay to discover available compute cycles on remote nodes. Our system exploits the *locality-awareness* property of Pastry [6] to maintain and locate nearby available resources to dispatch jobs for remote execution. 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. Messages are routed by Pastry to the destination node in $O(\log N)$ hops in the overlay network.

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 in each node n's Pastry routing table rows. Each such 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 only propagated to neighboring nodes within TTL hops in the overlay. Because 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 utilizes its accumulated knowledge about available resources to select a node for executing a job. Proximity and credit-worthiness of the remote node are taken into account during this selection process. The actual remote execution of the program and subsequent I/O activities are performed between the submitter node and the host node directly, and do not go through the overlay.

2.3. Host Safety and Portability through Java VM

Our cycle-sharing system uses a Java virtual machine as the host platform to execute submitted jobs, and programs submitted to the cycle-sharing system are Java programs. We chose a Java virtual machine as the platform because of Java's safety and portability properties.

Host safety is a key issue for the wide-spread use of cycle sharing systems. Untrusted submitter nodes may submit malicious applications into the cycle-sharing system to compromise the host machines that accept and execute the applications. A self-interested computation resource owner is only willing to join the cycle-sharing system to get benefits when hosting untrusted applications from the cycle-sharing system does not impose a security risk to the local computation resource, since this would be, in almost all cases, a much more significant cost than the benefits achieved. Furthermore, one of the goals of our system design is to allow cycle sharing with a minimum of human based administrative overhead. Therefore, the system needs to provide host security with a minimal involvement of local computational resource owner effort. 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. We note that any sandboxing mechanism providing isolation can achieve such a goal. For example, Tron [2] is an implementation of a process-level discretionary access control for Unix systems, and it allows users to specify the capability of a process's access to files on the machine. The FreeBSD 'Jail' [22] provides users 'root' privileges limited to the scope of the jail, allowing system administrators to delegate management capabilities for each virtual machine environment.

We use a Java virtual machine as the runtime environment in our current implementation, as Java's sandboxing mechanisms fit in well with the

above requirement. The Java 2 security design [19] enables fine-grained access, security checks for all Java programs and a configurable security policy. The host platforms in our cycle-sharing system uses Java virtual machines, whose inherent sandboxing and security design enables the protection of the host machines. By providing a central security policy applicable to each host platform, our system can obviate the necessity of setting up policies by each participant (therefore lowering the human intervention overhead): Each host machine simply configures the JVM according to the central policy; each job submitter must respect the central security policy while writing applications so as to make the applications runnable on remote sites within the system. If submitters have undergone 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.

Moreover, application portability is another key issue for the wide-spread use of cycle-sharing systems. Various modern hardware and operating systems expose heterogeneous application binary interfaces (ABI) to application developers. Again, our system needs to achieve portability without requiring a change of configuration of host machines, and to achieve it with low management overhead.

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 the bytecode specification is well defined. However, another important reason for its portability is that much of the functionality that is provided by system libraries (and that 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 submitter nodes to have a larger number of potential hosts to choose from, and increase the probability that a program will execute correctly on a remote host.

2.4. Fairness Through the Accounting System – A Brief Introduction

Fairness in our system is achieved through a credit accounting system. In the physical world, money is used as a conveyor of information about one's contribution to the economy; in our cycle-sharing system, credits are used to convey information about computational resource contributions to the system. The storage and retrieval of these credits is accomplished through the p2p network. Our credit mechanism provides a distributed, scalable system for making and accepting payments of resource usage, i.e., credits in the language of this paper. The details of the credit system will be discussed in detail in Section 3.

2.5. Progress and Correctness Monitoring – A Brief Introduction

GridCop is the component of our framework that allows the submitter to monitor a remotely executing application. The basic idea of GridCop is to instrument a program with beacons and to use beacon information to track the progress and verify the correctness of a remotely executing job. This technique treats a modified program control flow graph as a finite state automaton (FSA), and constructs a *transducer* using this FSA. The transducer is part of the program that is executed on the host, and emits beacon information that allows a corresponding FSA on the submitter machine to follow the progress of the job. If the beacon information follows legal transitions in the FSA on the submitter machine, execution correctness is verified and job progress as represented by the state transitions in the FSA is updated. Along with our credit system, progress information can be translated into an incremental payment according to any agreed upon scheme. The details of the GridCop system will be described in Section 4.

3. Accountability with the Credit System and Incremental Payment

We now describe the design of our credit system in detail.

3.1. Accounting through a Credit System

To ensure the compensation of consumed cycles consumed on node B by node A, we propose a distributed credit based mechanism. There are two building blocks of our approach: (1) Credits, which are digitally signed entities that can be 'traded' in exchange for resources, and (2) 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.

The distributed feedback database is built on top of the Distributed Hash Table (DHT) supported by the underlying structured p2p overlay for resource discovery described in Section 2.2. 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 to 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 2 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 2), A will issue a (digitally signed) credit to B (2 in Figure 2). This credit is similar to a claim in Samsara [8] – It can be 'traded' with other nodes for equivalent resources. The credit is labeled with a system-wide unique identifier which consists of A's identifier (e.g., A's IP address) and a unique sequence number – Unique in that A will issue no other credits with that number. B will digitally sign the credit and store it in a DHT-based repository (3 in Figure 2) by hashing the credit's system-wide unique identifier via a generic hash function, e.g., SHA-1.

Similarly, if B gives the credit to C (4 in the Figure), B will digitally sign the credit before giving it to C, and C will digitally sign it, compute the hash based on the credit's system-wide unique identifier (i.e. [identifier of A, credit sequence 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.

The approach guarantees that a transaction between two nodes is represented by a unique, unforgeable entity, which prevents a node forging

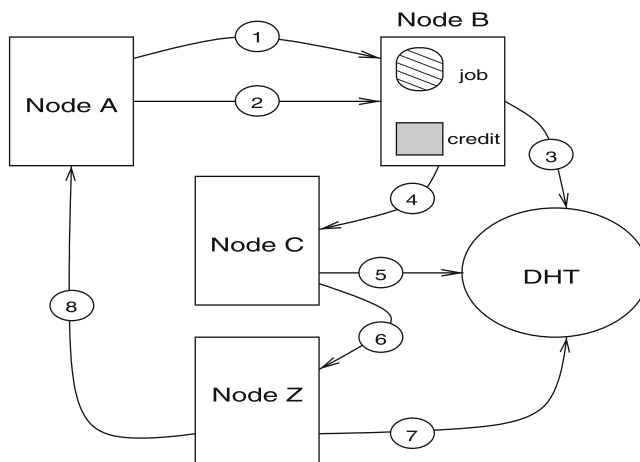


Figure 2 Various steps to ensure proper compensation of a contributed resource.

credits to deceive other nodes in the system. Additionally, by maintaining information about the current owner of a credit in the DHT, illegal replay or copying of a credit is also thwarted. 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) when it tries to insert the credit into the DHT, and Z could then refuse to run B's job. Should the DHT node on which the credit is saved be malicious and save both copies of the credit, a node checking on the credit worthiness of the issuing node can determine that there are two credits with the same identifier, and either ignore or factor this into its evaluation of both the issuing node *and* node B.

The feedback information is used to enforce contributions from selfish nodes as follows. In Figure 2, 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.

3.2. Incremental Payment

We assume in this project that we are not executing on truly malicious machines, rather we are running on machines that may be over-committed, or that may be 'fraudulently' selling cycles that do not exist in order to gain credits to purchase real cycles. Thus our system does not need to detect all fraudulent or over-committed systems, but rather must allow fraudulent and over-committed 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.

Similarly, one critical issue that needs to be solved is how to deal 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 learning of the completion of its remote job execution. To provide mutual assurances, we propose the issuing of incremental credits.

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 predetermined intervals. On each interval that A finds its job stalled, it calculates a probability $\mathtt{q}$, which increases exponentially with an increasing number of consecutive failures of B. A does not allow the job to continue further and issues a negative feedback for B with probability $\mathtt{q}$. 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.

Our GridCop monitoring component (see Section 4), along with the distributed credit system described in this section, enables incremental payment. This is especially important for a long running job which consumes significant computational resources.

4. The GridCop System

GridCop, the remote job progress and correctness monitoring system in our cycle-sharing system, enables a job submitter to periodically receive its submitted job's progress information and track progress and correctness. After verifying the correctness and progress of the program, the submitter can issue a credit payment (Section 3.2) to

the host for the consumed cycles that have been verified by the submitter.

Every participating node can submit jobs (i.e. be a *submitter* node) or host jobs (i.e. be a *host* node). A node can perform both roles simultaneously, i.e., its jobs can be running on remote nodes while jobs from other remote nodes are running on it. Before a program is executed, it is passed to our tool which transforms it into a program that executes on the host machine (*H-code*) and a program that executes on the submitter machine (*S-code*). The H-code is the original program augmented with *beacons* and auxiliary code that sends information about the program to the submitter machine. The S-code, executing on the submitter machine (or a trusted machine accessible from the submitter machine) uses this information to track the progress, and verify the execution, of the program.

Our GridCop system consists of compiler techniques to produce the H-code and S-code programs. The GridCop system includes the following three components: (i) A beacon message creation component on the host, which is integrated with the submitted application, (ii) a beacon reporting component on the host that sends beacon messages to the submitter, and (iii) a beacon message processing component on the submitter. The monitoring system architecture is shown in Figure 3. In the following, we first state the threat model assumed by our GridCop system, and then present the components of GridCop.

4.1. The Threat Model

Untrusted systems can be divided into two categories: Malicious and fraudulently irresponsible (e.g. the environment targeted by the Samsara [8] system.) Malicious systems are willing to expend significant resources to damage their victims. For example, a malicious system might be willing to execute a program until the final results are to be written to disk, and then terminate it. Our system is assumed to operate in a less hostile, but perhaps fraudulent environment. In particular, we assume that hosts are motivated by self-interest, that our programs execute on unmodified runtime system (such as JVMs), and that the source program is not altered. These are reasonable assumptions since runtime systems such as JVMs are large complicated pieces of software distributed in binary form, and are not easy to modify without access to the source code. Our monitoring system makes it very difficult to change a program without being caught. Moreover, because we *apparently* monitor almost the entire program execution (but *actually* monitor only a small part of the execution), fully automatic tools will not be effective. Program alteration requiring human intervention is costly enough to preclude its use.

4.2. Basic Beacon Scheme

Beacons emit information at significant program execution points. This information is sent to the

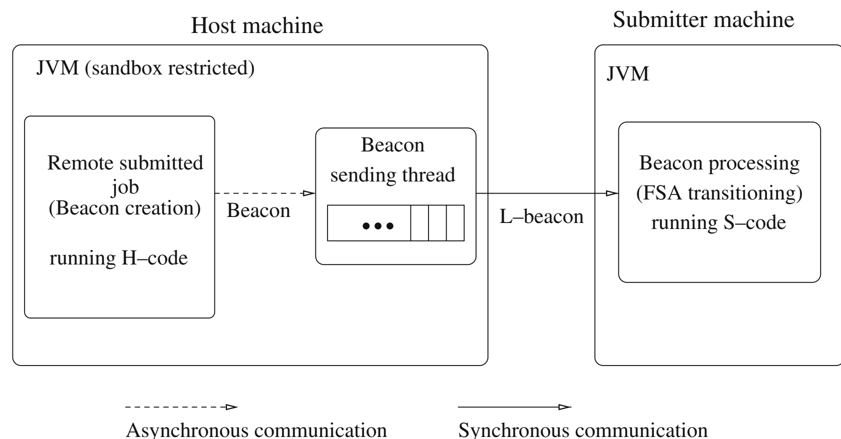


Figure 3 Components of the GridCop monitoring system.

submitter machine which uses it to determine what parts of the program have been executed. Placing beacons at every branch would provide very precise, fine-grained beacon information. Unfortunately, beacon processing would consume unacceptably large amounts of resources on the host and the submitter machines, since a large percentage of the instructions executed would be beacon related. Instead, GridCop places beacons at computationally significant points in the program.

For the H-code, the compiler finds computationally significant points in the program and places a beacon at each of them. It also inserts hard-coded routines into the code that aggregate and structure information provided by beacons, and that report the information to the submitter program. These routines are described in Section 4.4. An FSA is created in the S-code, where each state in the FSA corresponds to a beacon in the H-code. Beacon processing consists of checking if a sequence of beacon messages corresponds to legal transitions on the FSA. Viewed in this light, beacons in the H-code can be viewed as a transducer, i.e. a finite state machine that emits information on state transitions.

Because beacons report the progress of a job, the time between milestones should be reasonably long. Our approach is to place beacons at the beginning of method calls, but only at the beginning of method calls that perform a 'significant' amount of work. Two broad strategies can be used to determine methods with a significant amount of work: Profiling and compile time cost models. We have chosen to use a very crude compile-time cost model, namely, methods that contain loops are considered to be computationally significant. Future work for the project may involve refining this model (although for the current benchmark set, our current model is adequate). A beacon is inserted as the first statement in each computationally significant method. The beginning of the main program also gets a beacon, as does any node that can return from the program as the result of a normal program termination. For a multi-threaded program, each `run()` function is also treated as a main program.

As described above, when beacon information is sent to the submitter machine, the submitter checks the validity of the beacon information by traversing an FSA (multiple FSAs are traversed for a multi-threaded program, see details in Section 4.4). Each beacon inserted above corresponds to a state in the FSA, and the beacon's ID is the input for the transition to the state. The beacons inserted at the beginning and the end of the program serve as the start and the accept state, respectively, in the FSA.

4.3. Replay Resilient Beacon Scheme

Beacons are placed along certain control flow graph (CFG) edges, as described in Section 4.2, and identify the location of the program that is currently executing. Beacons provide fine-grained location information, but they are subject to replay and spoofing attacks if inserted in a deterministic way. A cheater can replay the stream beacons from a previous run for later requests to run the same job.

We prevent this type of attack by not always inserting beacons at the same locations when instrumenting a program (generating H-code). At each potential beacon site B, a beacon is inserted with probability P_B. If $P_B = 0$, no beacon is ever inserted at this site, if $P_B = 1$ a beacon is always inserted at this site. For $0 < P_B < 1$ a beacon may be inserted. By setting the values of P_B to be non-zero and less than one, each version of the program generated by our compiler will likely have a different set of beacons inserted and consequently a different sequence of valid beacon values. Because the values of P_B can be different at different candidate beacon sites B, the placement of beacons can be made more or less likely depending on the hotness of a program region. Attempts to replay the old beacon values will fail, with a high probability, because the replayed set of beacons will likely contain invalid beacon values. Because the bytecode for programs used in high performance computing are usually orders of magnitude smaller than the data they operate on, shipping (possibly) new bytecode with each execution imposes only a small overhead.

The algorithm of Figure 4 describes how to build an FSA with this replay resilient beacon generation scheme. We build the FSA for the submitter side in two steps. The first step transforms

Figure 4 Algorithm to Build an FSA for the Replay Resilient Scheme.

```
N_b ⇐ φ;
N_b̃ ⇐ φ;
for(each node n in G_F){
    if [(n meets beacon candidate condition) and (rand_withP_B(0,1) == 1)]
        N_b ⇐ N_b ∪ {n};
    else
        N_b̃ ⇐ N_b̃ ∪ {n};
}
while(N_b̃ ≠ φ){
    n_b̃ ← dequeue(N_b̃);
    for(each node n_p ∈ predecessor(n_b̃) ){
        for(each node n_s ∈ successor(n_b̃) ){
            Edges(G_F) = ⟨n_p, n_s⟩ ∪ Edges(G_F);
        }
    }
    Edges(G_F) = Edges(G_F) − {edge connected to N_b̃};
    N_b̃ ⇐ N_b̃ − {n_b̃};
}
G'_F ⇐ G_F;

State_FSA ⇐ {nodes in G'_F};
Transition_FSA ⇐ φ;
for(⟨n_i, n_j⟩ ∈ Edges(G'_F)){
    Transition_FSA ← Transition_FSA ∪
            {transition⟨n_i, n_j⟩ with ID n_j};
}
```

the control flow graph (*CFG*) G_F of the program into a graph G'_F that contains only nodes with *actually selected* beacons. First, we add nodes of G_F that contain potential beacon candidates (i.e. the computationally significant nodes), with probability of P_B to the set N_b. All other nodes (non-candidate nodes and not chosen beacon candidate nodes) of G_F are placed in the set $N_{\tilde{b}}$. For each node $n_{\tilde{b}} \in N_{\tilde{b}}$, edges are added between each of its predecessors $\{n_p\}$ and each of its successors $\{n_s\}$, i.e., edges $\langle n_p, n_s\rangle$ are added to G_F. Then all original edges connected to $n_{\tilde{b}}$, i.e., $\langle n_p, n_{\tilde{b}}\rangle$ and $\langle n_{\tilde{b}}, n_s\rangle$ in G_F, are removed from G_F. Finally, $n_{\tilde{b}}$ is removed from G_F. This procedure repeats till there exists no $n_{\tilde{b}}$ in G_F. We call the graph resulting from the above procedure G'_F.

The second step generates the FSA. The FSA transition table can be trivially constructed as follows. Let the set of states be the set of nodes in G'_F. For each edge $\langle n_i, n_j\rangle \in Edges(G'_F)$, there is a transition from n_i to n_j on the symbol which is the beacon ID of n_j. The start state of the FSA corresponds to the node at the beginning of the main program, and the accept states correspond to nodes that can return from the program.

Figure 5 shows a program fragment with beacons inserted with $P_B = 1$. For each computationally intensive function in Figure 5a, the compiler inserts a beacon instruction at the beginning of the function to emit a unique transition ID in Figure 5b. These IDs are treated as the transition symbols between states in the FSA. A transition ID always drives the FSA to the unique state named by the transition ID. Consider the FSA shown in Figure 5c: `bar2` in the example emits ‘3’, which causes a transition from any predecessor of `bar2`, in the FSA, to `bar2`.

4.4. Runtime Support for Beacons

At runtime, the beacon instructions inserted by the compiler are executed. Each beacon instruction generates a location ID (ID_L) and the current thread ID (ID_T), which together form a beacon message (ID_T, ID_L). For multi-threaded

Figure 5 A part of FSA derived from the Java Grande LU benchmark, $P_B = 1$.

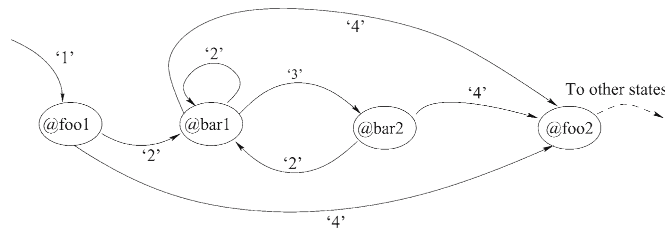
```
(in main function)
...
call foo1(...)
call foo2(...)
...


function foo1(...){
   for(...){
      call bar1(...)
      if(...) {
         call bar2(...)
         call lightbar(...) //this function is loopless
      }
   }
}
function bar1(...){ ... for( ...) ... }
function bar2(...){ ... for( ...) ... }
function foo2(...){ ... for( ...) ... }
```

(a) Pseudo code fragment from benchmark LU. All functions except lightbar() contain a loop

```
function foo1(...){ call create_beacon('1') ...}
function bar1(...){ call create_beacon('2') ...}
function bar2(...){ call create_beacon('3') ...}
function foo2(...){ call create_beacon('4') ...}
```

(b) Beacons added at the prologues of functions with loops



(c) Part of FSA corresponding to above program: transition symbols on the edges correspond to the unique IDs emitted by the inserted beacon instructions

programs, beacon messages from different *application* threads are multiplexed in the *beacon buffer* (using an adaptive scheme which will be described in detail next). ID_Ts are used to label different threads so that the beacon processing component on the submitter side can use them to track the progress of different threads. A different FSA is used for each application thread, thus ID_T in a beacon message identifies an FSA and ID_L drives transitions on that FSA.

The reporting component on the host is in a separate thread (*Sender*) that transmits beacon messages to the submitter through a TCP connection. The reporting component uses a *paced* beacon transmission scheme. A timed wait/notify mechanism, utilizing Java `wait(long)` and `notify()` functions, is used to achieve the paced transmission scheme as follows. *Sender* waits for an interval (set by the submitter when the program is submitted to the host) and wakes up to send the

messages in the beacon buffer. If the buffer gets full before a waiting interval passes, *Sender* is notified and wakes up to send the messages in the beacon buffer. The capacity of the beacon buffer is fixed, and it is set to contain 3,000 beacon messages in our current implementation. During the sending procedure, *Sender* copies the contents in the beacon buffer into a private buffer to clear the beacon buffer, and then leaves the critical section. It then sends the contents in the private buffer across the network. Thus the cross-network data transfer procedure is asynchronous to the application execution.

A beacon stride variable is used to achieve the adaptive beacon deposit scheme. The value of the stride variable controls the stride between two beacons, generated in an individual application thread, that is actually deposited into the buffer. Initially, the value of the stride variable is set to 1, i.e., every beacon message generated in an individual application thread will be deposited to the buffer. Whenever *Sender* sends the beacon buffer as a result of the buffer becoming full before the set interval passes, the value of the stride variable is doubled. For example, if the value of the stride variable is doubled from the initial value, every other beacon message generated in an individual application thread will be deposited to the buffer. When *Sender* sends the beacon buffer because the waiting interval passes, *Sender* checks the actual buffer usage ratio, i.e., the number of beacon messages that have been deposited into the buffer over the buffer size. The stride variable value is decreased to half if the ratio is less than 0.5. In this way, the rate that beacon messages are deposited in the beacon buffer is set adaptively: When the beacon messages are generated at a high volume, the stride distance between two beacons that are deposited into the buffer is increased; when the beacon messages are generated at a low volume, the stride distance is decreased.

The submitter node uses the beacon messages sent by the host as input to the FSAs to track the progress of the remote job. The stride value and the number of beacon messages contained in each beacon buffer that is sent from the host machine is located at the beginning of each message stream. The submitter de-multiplexes and processes the received beacon messages in a beacon buffer as follows. Upon receiving a beacon buffer, the submitter first looks at the value of the stride variable, n, to see the stride value for the beacon messages in this beacon buffer. It then reads each beacon message, locates an FSA using ID_T and drives transitions using ID_L. Each beacon value in the buffer is processed by comparing it to states exactly n steps next from the current state by performing a lookup in the FSAs transition table. If the beacon value does not match a valid transition from the current state, it is an illegal transition and the appropriate action is taken. Because the beacon message strides are changed adaptively so that the number of beacon messages is limited, the submitter overhead to process these beacons is also limited. Buffers continue to be received, and beacon values in the buffers continue to be processed, until the submitter receives the final state beacon value.

4.5. Experimental Results

In this section, we present experimental data on the monitoring overhead incurred by GridCop and the simulation results showing the effectiveness of the monitoring mechanism.

4.5.1. *Experimental Platform*

Our experiment was run on a submitter/host pair located at University of Illinois at Urbana-Champaign and Purdue University. The submitter machine, located at UIUC, is an uniprocessor with an Intel 3 GHz Xeon processor with 512 KB cache and 1 GB main memory. It runs the Sun JDK 1.5.0 and the Linux 2.4.20 kernel. The host machine located at Purdue is a Dell PowerEdge SMP server with 4 × 1.5 GHz Intel Xeon processors, each with 512 KB cache and sharing 4 GB main memory. It runs the Sun JDK 1.4.2 and the Linux 2.4.20 kernel. Both machines are connected to the campus networks.

4.5.2. *Benchmarks and Configuration*

As our application benchmarks we use the parallel Java Grande benchmark suite version 1.0 [35],

a standard benchmark suite for computationally intensive Java applications. Suite II of the parallel Java Grande benchmarks contains simple kernels which are commonly found in the most computationally intensive parts of real numerical applications. The Java Grande benchmarks are self-initializing, i.e., there is no network activity to send program data sets. We used data size B as the input to our experiments. We implemented the runtime support library described in Section 4.4, and application programs were hand-transformed using the techniques described Section 4.2.

In our measurements, we actually inserted beacons at each potential beacon candidate site, which represents a scenario of $P_B = 1$ described in Section 4.3. The inter-transmission intervals of the beacon reporting component were set to 2 s. This is a highly aggressive monitoring scenario. In an actual system, the inter-transmission interval would be in tens of seconds or minutes. Considering the above two configurations, our experiment actually provides an upper bound of the performance overhead and network traffic incurred by using our monitoring system.

4.5.3. *Runtime Computation Overhead*

To simulate long running jobs, a loop is added outside the individual kernels in the benchmarks. Each benchmark was run in 1, 2 and 4-thread mode to evaluate the scalability of our system design by showing the system performance overhead with different degrees of parallelism.

On the host side, we first measure the time to run the original benchmarks on our host machine, which reflects the scenario of remote job execution without monitoring. These form our baseline numbers for host performance. We then run the manually transformed H-code and S-code versions of the same benchmarks on the submitter/host pair, which reflects the scenario of a remote job submission with monitoring. Figure 6 shows the overhead of job executions with beacons over the corresponding un-monitored baseline job execution times. We observe that the overhead increases as the degree of parallelism increases. This is because the higher the degree of parallelism is, the more expensive it is to synchronize the beacon depositing thread and the main computation threads. Our experimental results show that the performance overhead is under 4.5%.

On the submitter side, we measured the time to execute the benchmarks on the submitter in the single-thread mode and used it as the submitter baseline (as the submitter is not an SMP machine). We then measured the CPU time used to process the beacons while monitoring remote benchmarks running in 1, 2 and 4-thread modes. These are the computation resource costs on the submitter. Figure 7 shows the ratio of CPU time used to monitor a benchmark, which include the time used to receive the beacon packets and process them, over that of running the same benchmark in the single-thread mode locally on the submitter. Note that we used the system ‘time’ command to measure the submitter’s system and user time

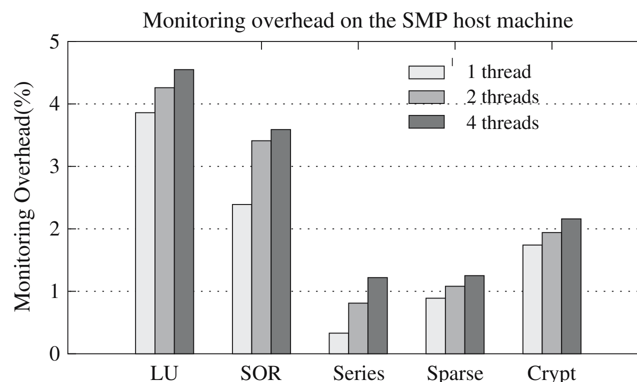


Figure 6 Overhead of executing and transferring beacons.

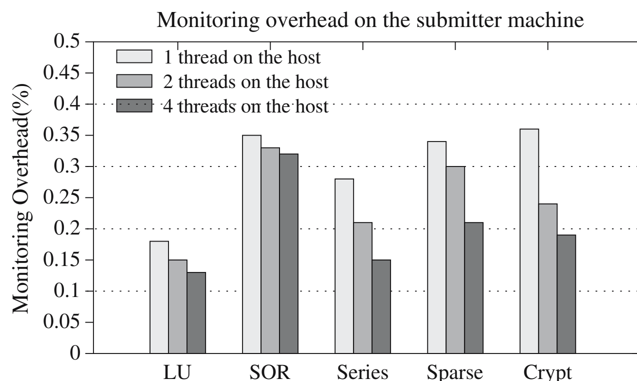


Figure 7 Submitter's computational cost of monitoring a remote job over running it locally.

used by the monitoring process. This measurement includes the JVM startup cost, which makes the submitter cost not proportional to the number of the beacon messages received and processed. However, this does give us a sense how effectively the computation has been outsourced. We observe that the ratio of CPU usage for monitoring over locally executing the benchmark is always under 0.4%, which reflects the fact that the submitter effectively outsources the computation burden to the host machine. Another point deserves an explanation in Figure 7. When the host runs with a higher degree of parallelism, beacon messages from more threads multiplex together and the beacon buffer is filled at a higher speed. Our '*adaptive beacon deposit scheme*' (see Section 4.4) increases the stride value and skips beacons when necessary. Therefore, the number of the beacon messages transferred per unit time is not changed significantly. As the execution time is shorter when running a benchmark with a higher degree of parallelism, the number of beacons messages transferred is less. For example, the numbers of total beacon messages received and processed at the submitter side when the LU benchmark was running in 1, 2 and 4-thread mode are 894,390, 665,852 and 566,260, respectively. This explains why the submitter cost decreases when the degree of parallelism increases.

4.5.4. *Network Bandwidth Overhead*

Another important metric to evaluate our system is its network resource usage. Since network resources are finite, it is necessary to limit the amount of data sent from the host machine to the submitter machine. To see how effectively we accomplished this, we measured the actual network traffic incurred by our beacon subsystem for each benchmark under different execution modes.

The capability to serialize objects is one of the features of Java that simplified our implementation and increased its robustness. Beacons are placed in serializable objects to make the submitter's understanding of the representation of data from the host easier. However, this mechanism transfers more data across the network than just sending the raw data. To measure the actual network traffic sent by our system, we serialize an object to be sent across the network by the host into a ByteArrayOutputStream object. We then measured the actual transferred size of the serializable object as shown in Figure 8.

```
...
ByteArrayOutputStream baos =
        new ByteArrayOutputStream();
ObjectOutputStream objectos
        = new ObjectOutputStream(baos);

objectos.writeObject(serializable_object_to_send);
//serializable_object_to_send contains beacons

byte[] bytes = baos.toByteArray();
objectos.close();
baos.close();

size_sum += bytes.length;
//size_sum will provide network traffic information
...
```

Figure 8 Code to measure the network traffic caused by our approach.

Table 1 Average network traffic during execution monitoring.

	1 thread	2 threads	4 threads
LU	8.4 KB/s	8.7 KB/s	9.1 KB/s
SOR	2.5 KB/s	3.2 KB/s	3.4 KB/s
Series	75 Byte/s	77 Byte/s	78 Byte/s
Sparse	78 Byte/s	81 Byte/s	94 Byte/s
Crypt	79 Byte/s	85 Byte/s	100 Byte/s

We measured the network traffic from beacons for different benchmarks running with different numbers of threads. Table 1 shows the highest average network traffic per unit time for our benchmarks is only 9.1 KB/s, showing that our monitoring mechanism consumes a very small amount of bandwidth.

4.6. Effectiveness of Cheating Node Detection

In this section, we analyze the effectiveness of our GridCop remote job monitoring system to catch fraudulent hosts and enforce a fair and efficient cycle-sharing system.

4.6.1. *Probability of Successful Replay Attack*

For simplicity, we analyze the case where a fraudulent host records the beacon stream while running the job, and replays it to the submitter when asked to run the application again. Here we denote the number of potential beacon sites as N, and we denote the probability that each potential beacon site B is actually marked as beacon, i.e., an beacon instruction is inserted at B, as P_B. The probability that any two versions of the generated H-codes get identical beacons generated at a specific potential beacon site, i.e., the probability that both have a beacon generated or neither has a beacon generated, is $P = (P_B)^2 + (1 - P_B)^2$. Therefore the probability that two versions of the generated H-codes are identical is $P_{suc} = ((P_B)^2 + (1 - P_B)^2)^N$. We assume that there are N potential beacon sites in the code region of 'possibly executed' during runtime, i.e., code not within exception handling functions, or dead code region. P_{suc} represents the probability of a successful replay attack.

As long as the submitter selects P_B with a value not too far away from 0.5, the probability of a successful replay attack is extremely low. Figure 9 shows the successful replay attack rate if a submitted job has $N = 20$ potential beacon sites. For example, if the submitter chooses $P_B = 0.7$, the successful rate of a replay attack is about 10^{-5}; and if the submitter chooses $P_B = 0.5$, the successful rate of a replay attack is 10^{-6} or so.

The probability of a successful replay attack, when the submitter chooses P_B as a certain value decreases quickly as N increases. Figure 10 shows different successful replay attack rates when a job has different numbers of potential beacon sites N when the submitter chooses $P_B = 0.5$. We note

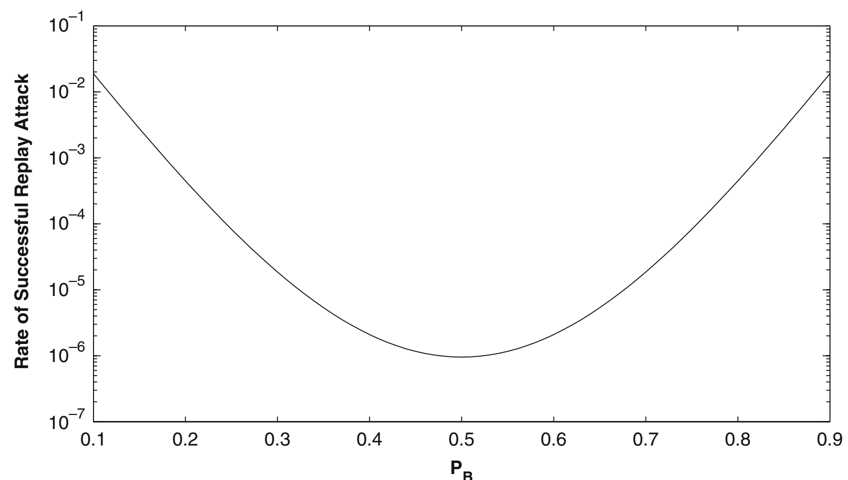


Figure 9 Probability of successful replay attack vs. P_B (when $N = 20$).

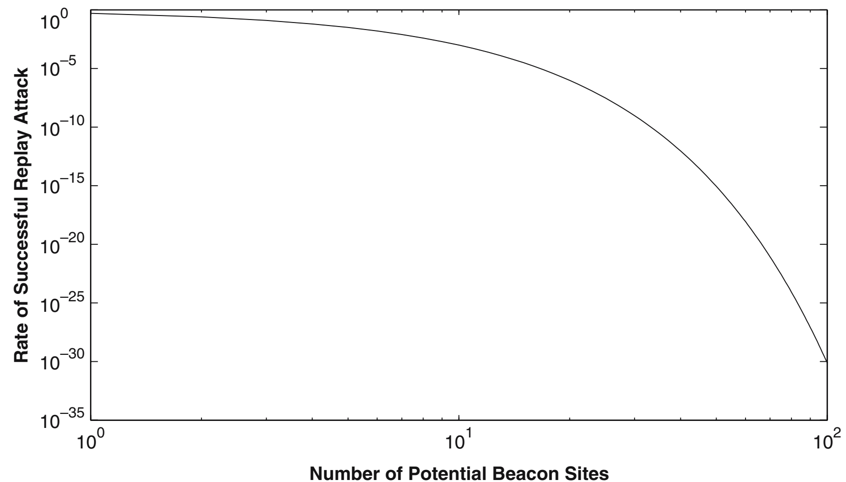


Figure 10 Probability of successful replay attack vs. N ($P_B = 0.5$).

that, the number of potential beacon sites is large for 'real life' computations.

Both above figures give us the analytical proof that our replay resilient beacon mechanism can detect cheating nodes with high fidelity.

4.6.2. *Simulation of the System Usage with Fraudulent Nodes Detected*

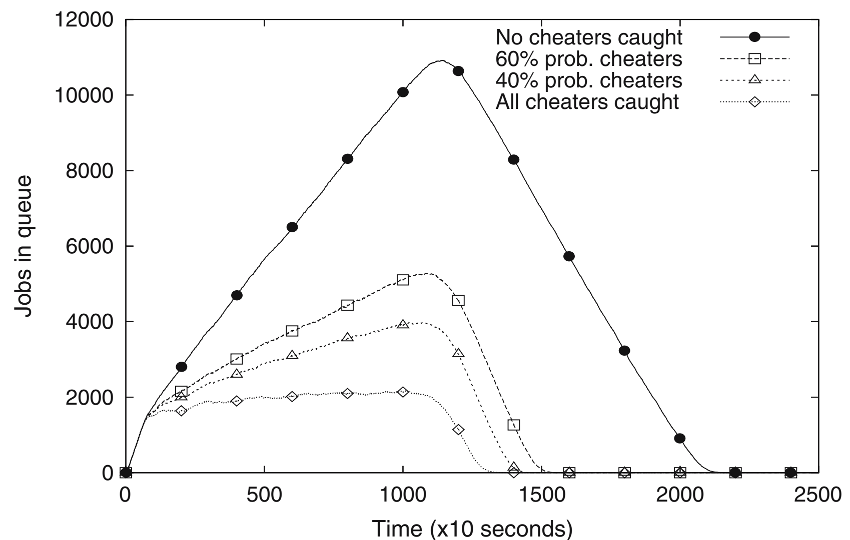
Now we simulate the cycle-sharing system with the functionality of detecting cheating nodes with GridCop and isolating those nodes from the system. The non-cheating nodes faithfully run the jobs sent to them, and the cheating nodes either refuse to run the jobs even if they are not busy, or abandon a running job before completion. For the purpose of our experiment, we simulated a pool of 1,000 host nodes.

To drive the simulation we created a job trace as follows. We selected 100 application executions uniformly randomly from the 15 different application executions shown in Figure 6. Next, we determined the mean execution time (T_μ) for the 100 selected jobs and created a random job issue sequence such that the inter-arrival time between two consecutive jobs has a uniform random distribution with a means of T_μ. 500 of the 1000 nodes issued 100 jobs each using the created trace. We made 250 of these 500 nodes cheat. The presence of these nodes affects the time it takes for jobs issued by non-cheating nodes to complete.

Figure 11 shows the number of jobs issued by non-cheating nodes only that are issued but waiting in queue over time. Note that the total number of jobs issued by non-cheating nodes is 25,000. The topmost curve shows the scheme when no cheaters are caught. Here jobs from non-cheating nodes have to compete with jobs from cheating nodes, and hence it takes longer for them to complete. When a job is sent to a cheating node, it can either immediately refuse to run the job or abandon the job without completing it. In either case, the submitter detects that its job is not running and resubmits it to some other (probably non-cheating) node. The process is repeated until the job successfully completes.

The bottom curve in Figure 11 shows an ideal situation where all cheaters are known a priori. In this case, the non-cheating nodes send their jobs to other non-cheating nodes only, and do not run any jobs submitted by a known cheating node. The number of jobs enqueued and submitted by non-cheating nodes increases little due to the issue sequence, and all jobs complete within 2,000 s after the issuance of the last job at 11,730 s. The next two curves show more realistic cases where cheaters accept a job, but abandon it after processing it for a random time. We assume that within 2 s from a node abandoning a job, i.e., the next beacon interval, the cheating node is caught. We simulate two kinds of cheating nodes: a) Cheaters that simply refuse to run jobs and can be detected immediately; and b) cheaters that

Figure 11 Number of jobs issued but not completed over time. The percentages represent the portion of cheaters that are not immediately caught.



abandon the execution of an accepted job after running it for a random amount of time from the beginning of the job to the finishing time of the job (i.e.,$0 < T_{abandon} < T_{job}$). The percentage in the legend represents the percentage of the total cheating nodes that exhibit behavior (b), the rest of the cheating nodes exhibit behavior (a). The curves show that our scheme is able to catch the cheaters, and the jobs for non-cheating nodes complete much faster than the case where cheaters were not caught at all. In this scenario both probabilistic cheater schemes were able to complete under 15,560 s, compared to 21,690 s when no cheaters were caught. In contrast, the ideal case took 13,730 s to complete all the issued jobs. The simulation shows that our scheme is effective in determining the cheating nodes, and isolating their effect from the overall system.

5. Related Work

In this section, we present the related research and the difference between them.

5.1. Cycle Sharing Over the Internet

The idea of cycle sharing among a large number of administratively independent, geographically dispersed, off-the-shelf desktops was popularized by the SETI@home [34] project. Similar approaches for solving large scale scientific problems are also adopted in systems such as Distributed.NET [11], Entropia [13], Genome@home [17], 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 [4]. 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, thereby increasing the instantaneous compute capacity of the system.

Various Grid platforms also share the same goal of distributed sharing resources. Condor [25] provides a mechanism for sharing resources in a single administrative domain by harnessing the idle-cycles on desktop machines. Globus [14] 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 [4]. PUNCH [24] decouples the shared resource users from the users on the underlying operating system on each resource, hence eliminating the need for accounts on all the shared resources. The Sun Grid Engine [37] is another system that harnesses the compute powers of distributed resources to solve large scale scientific problems. 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.

We proposed the self-interest motivated cycle-sharing system in [5]. In that paper, we proposed the system design and described the feasibility of a remote computation monitoring system. In this paper, we present in detail the prototype of a remote monitoring system, GridCop.

5.2. Progress Monitoring and Debugging

Various remote debugging techniques have existed for years, see [7, 30]. Remote debugging techniques are used as a development facility to help developers to find bugs on a *trusted* remote platform during the program development phase. Our approach differs from these techniques in that our approach collects execution correctness information on an *untrusted* platform after the program has been developed and released, and in how the data is gathered.

Program monitoring is also employed in the Globus project for providing better quality of service [15]. This monitoring is either achieved indirectly by determining the resource utilization of the program, or by modifying the program to insert explicit calls to the Globus API. The motivation of our work is different in that we are using the monitoring to determine if we are getting a resource as promised.

With the increasing popularity of volunteer based cycle sharing, efficient protection against malicious machines has become an important research topic. Sarmenta [33] discussed a spot checking mechanism to catch malicious machines (saboteurs). Du et al. [12] proposed a Merkle (Hash) tree based technique to detect cheating nodes when embarrassingly parallel computations are being performed. Both of the above techniques ensure the integrity of participant machines by checking a subset of *independent computations* completed by the participant machines. Over time, our approach monitors the correctness of all parts of an application. Moreover, our technique monitors the progress of the application, enabling partial payments or detection of errors before a long running application has finished.

We studied progress monitoring of jobs running on potentially fraudulent hosts in other projects [5, 39]. We proposed the idea of using a finite state automaton to monitor a remote computation in [5] and investigated the feasibility of such idea, and we also recognized the potential replay attack. We proposed a solution to the replay attack in [39] by partially verifying the computation. Specifically, the inputs and outputs of randomly selected computation regions (which we called recomputation beacons) are transferred back to the submitter, and the submitter gets local outputs by recomputing the inputs to check if the remote outputs are correct. In this paper, we solve the replay attack problem by probabilistically generating beacons at potential beacon sites. Thus we eliminate duplicate computation on the submitter side. This approach significantly reduces the network bandwidth usage of the GridCop system, as we do not transfer the data associated with recomputation beacons. Our difference from [39] lies also in the runtime system support. We adaptively adjust the beacon stride when the beacon buffer gets full rather than simply dropping the beacons. This scheme makes the beacon message tracking across different beacon buffers more accurate.

5.3. Fair Peer-to-Peer Resource Sharing

There is a large body of work on enforcing fairness in resource sharing [3, 8, 9, 16, 31] and on creating incentives for fair sharing [18, 21, 38] in peer-to-peer systems. Our contribution is orthogonal to this previous work in that our focus is on monitoring job progress in the context of cycle sharing in the Internet, such as in peer-to-peer systems.

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 [16] 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 means for advance resource reservations.

There have also been efforts to design a general framework for trading resources in p2p systems. *Data trading* [3] is proposed to allow a consumer and a resource provider to exchange an equal amount of data, and cheaters can be punished by withholding their data. The approach requires symmetric relationships and is not directly applicable to p2p systems where there is little symmetry in resource sharing relationships. The use of micro-payments as incentives for fair sharing is proposed in [18]. Fileteller [21] suggests the use of such micro-payments 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, but a node verifies the credit reports of a remote node only when it has to do an exchange with it, which is a direct benefit.

OurGrid [1] uses an autonomous reputation scheme, called the Network of Favors to discourage free riding in a peer-to-peer CPU-sharing Grid. In particular, donating a resource is a favor, and each autonomously prioritizes peers who have reciprocated more favors in the past. In contrast, our credit system does not directly depend on the pair-wise favors and is more flexible in encouraging resource sharing. BOINC [10] is a volunteer computing system and provides a mechanism called 'trickle messages' to convey credit and report computational status which permits incremental credits for long running jobs. Similar to BOINC, our credit system, along with our GridCop monitoring system, also allows incremental payment. Different from BOINC, our credit system manages credits in a distributed manner by building on top of a DHT.

6. Conclusion

We have described a cycle-sharing system motivated by participants' self-interest in a large scale, and we have presented our solutions to the main challenges that obstruct the deployment of such a system. We believe this is the first comprehensive solution to assure system fairness, protect not only host machines' interests (e.g., safety) but also submitter machines' interests (e.g., correctness assurance), and bound participants' risks using the incremental payment mechanism. Moreover, the overhead of performing monitoring between the job submitter and the job host is shown to be low, both on the host side and on the submitter side. This technique opens the way for exploiting idle cycles across the Internet in a dynamic, decentralized and accountable fashion.

References

1. Andrade, N., Brasileiro, F., Cirne, W., Mowbray, M.: Discouraging free-riding in a peer-to-peer Grid. In: Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC13), June 2004

2. Berman, A., Bourassa, V., Selberg, E.: TRON: Process-specific file protection for the UNIX operating system. In: Proceedings of the USENIX 1995 Technical Conference, New Orleans, Louisiana, January 1995
3. Cooper, B.F., Garcia-Molina, H.: Peer-to-peer resource trading in a reliable distributed system. In: Proc. First International Workshop on Peer-to-Peer Systems, 2002
4. Butt, A.R., Adabala, S., Kapadia, N.H., Figueiredo, R.J., Fortes, J.A.B.: Grid-computing portals and security issues. Journal of Parallel and Distributed Computing: Special issue on Scalable Web Services and Architecture **63**(10), (October 2003)
5. Butt, A.R., Fang, X., Hu, Y.C., Midkiff, S.: Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing. In: Proceedings of the 3rd USENIX Virtual Machines Research and Technology Syposium (VM'04), May 2004
6. Castro, M., Druschel, P., Hu, Y.C., Rowstron, A.: Exploiting network proximity in peer-to-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002. http://research.microsoft.com/~antr/PAST/localtion.ps (17 Oct 2003)
7. Cheng, D., Hood, R.: A portable debugger for parallel and distributed programs. In: Proceedings of the 1994 ACM/IEEE conference on Supercomputing (SC'94), November 1994
8. Cox, L.P., Noble, B.D.: Samsara: Honor among thieves in peer-to-peer storage. In: Proc. 19th ACM Symposium on Operating Systems Principles, October 2003
9. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: Proc. SOSP, October 2001
10. David, A.P.: BOINC: A system for public-resource computing and storage. In: Proc. 5th IEEE/ACM International Workshop on Grid Computing, November 2004
11. Distributed.net. distributed.net projects (11 April 2003). http://www.distributed.net/projects.php (28 September 2003)
12. Du, W., Jia, J., Mangal, M., Murugesan, M.: Uncheatable Grid computing. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), March 2004
13. Entropia, Inc. {Entropia: PC Grid Computing (16 June 2003). http://www.entropia.com/index.asp (28 September 2003)
14. Foster, I., Kesselmann, C.: Globus: A metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. **11**(2), (Jan. 1997)
15. Foster, I., Roy, A., Sander, V.: A quality of service architecture that combines resource reservation and application adaptation. In: Proc. 8th International Workshop on Quality of Service, June 2000
16. Fu, Y., Chase, J., Chun, B., Schwab, S., Vahdat, A.: SHARP: An architecture for secure resource peering. In: Proc. 19th ACM Symposium on Operating Systems Principles, October 2003
17. Genome@home.Genome at home. http://www.stanford.edu/group/pandegroup/genome/index.html (December 16, 2004)
18. Golle, P., Leyton-Brown, K., Mironov, I.: Incentives for sharing in peer-to-peer networks. In: Proc. Third ACM Conference on Electronic Commerce, 2001
19. Gong, L.: Inside Java2 Platform Security. Addison-Wesley, 1999
20. Grimshaw, A.S., Wulf, W.A.: Legion – A View from 50,000 feet. In: Proc. 5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96), 1996
21. Ioannidis, J., Keromytis, A., Prevelakis, V.: Fileteller: Paying and getting paid for file storage. In: Proc. Sixth Annual Conference on Financial Cryptography, 2002
22. Kamp, P.-H., Watson, R.N.M.: Jails: Confining the omnipotent root. In: Proceedings of SANE 2000 Conference, May 2000
23. Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload Management with LoadLeveler. IBM International Technical Support Organization, 2001. http://www.ibm.com/redbooks (Dec. 17, 2004), publication number SG24-6038-00
24. Kapadia, N.H., Fortes, J.A.B.: PUNCH: An architecture for Web-enabled wide-area network-computing. Cluster Computing: The Journal of Networks, Software Tools and Applications **2**(2), (Sep. 1999)
25. Litzkow, M., Livny, M., Mutka, M.: Condor – a hunter of idle workstations. In: Proc. 8th International Conference on Distributed Computing Systems (ICDCS 1988), June 1988
26. Moreira, J., Midkiff, S., Gupta, M., Artigas, P., Wu, P., Almasi, G.: The NINJA project: Making Java work for high performance computing. Commun. ACM **44**(10), (October 2001)
27. Moreira, J.E., Midkiff, S.P., Gupta, M.: From flop to megaflops: Java for technical computing. ACM Trans. Program. Lang. Syst. **22**(2), (March 2000). IBM Research Report RC 21166
28. Nile. Scalable Solution for Distributed Processing of Independent Data. http://www.nile.cornell.edu/index.html (September 29, 2003)
29. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proc. ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'01), 2001
30. Redell, D.D.: Experience with topaz teledebugging. In: Proceedings SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, May 1988
31. Rowstron, A., Druschel, P.: PAST: A large-scale, persistent peer-to-peer storage utility. In: Proc. 18th ACM Symposium on Operating Systems Principles, October 2001
32. Rowstron, A., Druschel, P.: 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), November 2001
33. Sarmenta, L.F.: Sabotage tolerance mechanism for volunteer computing systems. In: CCGrid'01, May 2001
34. Seti@home. Search for extraterrestrial intelligence at home. http://setiathome.ssl.berkeley.edu/index.html (December 16, 2004)

35. Smith, L.A., Bull, J.M., Obdrzalek, J.: A parallel java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001), November 2001
36. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: 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), 2001
37. Sun(TM) Microsystems. Sun ONE Grid Engine Software (26 June 2003). http://www.sun.com/software/gridware/sge.html (29 September 2003)
38. Ngan, T-W.J., Wallach, D.S., Druschel, P.: Enforcing fair sharing of peer-to-peer resources. In: Proc. Second International Workshop on Peer-to-Peer Systems, 2003
39. Yang, S., Butt, A.R., Hu, Y.C., Midkiff, S.P.: Trust but verify: Monitoring remotely executing programs for progress and correctness. In: Proc. of PPOPP'05, June 2005
40. Zhao, B.Y., Kubiatowicz, J.D., Joseph, A.D.: Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB-CSD-01-1141, U. C. Berkeley, April 2001