Fault Tolerance in Ginger

João Paulino
joaopaulino@ist.utl.pt

Instituto Superior Técnico
INESC-ID

Abstract. The use of several machines to speed-up the execution of long running tasks has firstly emerged in institutional environments. Grid Computing systems schedule jobs to be executed in idle workstations owned by one or multiple institutions. During the 90's with the growth of the Internet came the possibility of using the spare cycles of personal computers owned by the general public (Public Computing). However, the great majority of the deployed systems focus on the execution of research applications and only allow their users to donate CPU cycles to a given cause (e.g., Seti@home client enables the general public to analyse radio signal for SETI). Ginger (Grid Infrastructure for Non Grid EnviRonments) proposes a system that enables common users to directly exchange processing power among themselves in order to speed-up the execution of common applications (e.g., video compression, photo-processing, ray-tracing). This third party execution environment is characterized by highly transient participants and possibly less reliable hardware. This work explores checkpoint restart mechanisms and result verification techniques to improve fault tolerance in Ginger.

Keywords: fault tolerance, result verification, checkpoint restart

1 Introduction

The execution of long running applications has always been a problem. Even with the latest developments of faster hardware, the execution of many long running algorithms is still infeasible by common computers, for it would take months or even years. Even though super-computers could speed up these executions to days or weeks, almost no one can afford them. The idea of executing these in several machines was firstly explored in networks or clusters of workstations [37, 4], using dozens of dedicated homogeneous machines locally interconnected. Later, grid computing systems explored the opportunistic use of hundreds of heterogeneous machines owned by institutions. Most recently, with public computing systems [3, 21] it became possible to harvest spare CPU cycles present in thousands of machines owned by the general public. Many public computing projects have been successful, and shown that the general public is willing to donate their CPU cycles to global causes. However, no project has successfully enabled the users to speed up their long running applications. Ginger proposes to fill this gap by merging grid computing, peer-to-peer and public computing.

1.1 Grid Computing

Grid computing aims at providing a virtual super-computer with increased capabilities at a low cost. These systems are composed by well managed hardware (e.g., workstations) owned by multiple institutions geographically distributed around the Globe. Mutka and Livny [29] studied the patterns of activity of institutional workstations and observed that they are idle up to 70% of the time. Grid computing makes good use of these idle cycles providing a high performance execution environment with no perceptive additional costs. Projects like Globus [17] and MyGrid [10] have studied the implementation of grid infrastructures that provide high-level meta-computing services enabling the efficient development of applications. The Condor project [23] takes advantage of CPU cycles present in idle workstations to speed up the work on the busiest ones.

1.2 Peer-to-peer

Peer-to-peer systems do not have a widely accepted definition. Many definitions can be found, Clay Shirky [36] wrote:

> *"An application is peer-to-peer if it aggregates resources at the network's edge, and those resources can be anything. It can be content, it can be cycles, it can be storage space, it can be human presence."*

Therefore, peer-to-peer systems are used to share resources like memory, CPU, storage, bandwidth and even human presence between peers located at the edges of the Internet. The majority of the definitions agree on this point, the hot topic of discussion is the architecture. Considering the Client Server model entities, the client and the server, we can define a peer saying that it implements the functionalities of both client and server. In some systems, the peers rely on a central server for support services (e.g., bootstrapping) or even for some basic operation functions (e.g., indexing and searching). Some authors argue that there can be some central entity, though the sharing of resources must be done directly between peers, others argue that in peer-to-peer there is no central entity whatsoever. Apart from the definition, peer-to-peer systems are characterized by their scalability, their ability to adapt to failures and their capability of accommodating highly transient node populations while maintaining acceptable connectivity and performance.

1.3 Public Computing

Public computing stems from the fact that the World's computing power and disk space is no longer exclusively concentrated in supercomputer centres and machine rooms. Instead, it is distributed in the hundreds of millions of personal computers and game consoles belonging to the general public. By combining ideas from grid computing and peer-to-peer systems it is possible to take advantage of idle resources of personal computers. Public computing emerged in the mid-90's with projects like distributed.net [11] and GIMPS [18]. While grid makes use of machines that 1) are managed by IT professionals, 2) have up-times of 24 hours per day and 3) are trustful, public computing faces

issues of malicious behaviour, highly transient populations and users without enough IT know-how. These issues make public computing a whole new challenge.

Public computing projects so far focus on mankind-related causes: Seti@home [3] analyses radio signal trying to find evidence of extraterrestrial life; Folding@home [21] searches for the cures of diseases like cancer, Parkinson's, Alzheimer's, etc. studying how proteins fold; distributed.net [11] solves brute-force cryptographic challenges exposing vulnerabilities; GIMPS [18] searches Mersenne prime numbers.

In order to motivate public to donate their spare cycles to such causes these projects usually have public ranking tables or in some cases money prizes. Public computing projects have attracted great attention from the general public, communities have been created around the projects showing that the volunteers are willing to actively participate in these projects. David P. Anderson [1] claims that it is expected that in the future many more research projects will take advantage of volunteer execution and people will have to choose which projects are worth to consume their cycles, these choices can condition the evolutionary path of science in a democratic way.

Nonetheless, none of the high-performance computing systems developed so far enables the general public to run their common desktop applications faster.

1.4 GINGER (Grid Infrastructure for Non Grid EnviRonments or GiGi)

GINGER [40] proposed a network of favours where every peer is able to submit his work-units to be executed on other peers and execute work-units submitted by other peers as well. GiGi combines institutional grid infrastructures, distributed cycle sharing and decentralized Peer-to-peer architectures. GiGi is able to run unmodified common desktop applications, however not all applications are fit for distributed computing:

> *"To be amenable to public computing, a task must be divisible into independent pieces whose ratio of computation to data is high (otherwise the cost of Internet data transfer may exceed the cost of doing the computation centrally)."* [1]

In order to be able to run an interesting variety of applications GiGi proposes the concept of Gridlet, a semantics-aware unit of workload division and computation off-load (basically the data, and the code or a reference to it). Therefore, GiGi is expected to run applications like audio and video compression, signal processing related to multimedia content (e.g., photo, video and audio enhancement, motion tracking), content adaptation (e.g., transcoding), and intensive calculus for content generation (e.g., ray-tracing, fractal generation).

The rest of this report is structured as follows. The next section describes the objectives of this work in the area of fault tolerance of GiGi. Section 3 makes an overview of the work done in the area. Section 4 proposes a fault tolerant solution. Section 5 explains how the work will be evaluated. Section 6 concludes.

2 Objectives

This work intents to provide fault-tolerance in Ginger. It will explore: techniques for verification and validation of the returned results; checkpointing mechanisms to mini-

mize the loss of already performed work when a peer fails; and mechanisms to monitor the evolution of submitted jobs.

2.1 Result Verification and Validation

The returned results may be wrong due to malicious/cheating behaviour or due to the occurrence of a fault. In order to discard bad results, every received result must be checked. Two techniques will be implemented to confirm the received results: validation[1](testing if the result makes sense) and verification[2] (testing if it is the accurate result of that specific job). Beside the results, the claims of costs should also be validated.

Result validation confirms if the result is in the expected format. If it is not a previous result or just random bytes. This light technique will be able to discard some bad results with very low effort. Though, it is a very limited technique. The validated results should be further verified.

Result verification is a more expensive technique that will ensure that a result is the correct for that execution. This is usually implemented using voting quorums, requiring the redundant execution of every work unit. Even though it is not very efficient, it is quite effective when not dealing with malicious collectives.

2.2 Checkpointing

The majority of the deployed high performance computing systems implement checkpoint restart mechanisms. Checkpointing is the process of saving a running application state to stable storage (e.g., to a file in the local disk). This file can be used later to resume the application's execution if a failure occurs. This fault tolerance mechanism is used to minimize the loss of results of already performed execution when a peer fails, allowing the execution to be continued from the point where it was saved.

In GiGi we want to keep track of the progress of a gridlet. If a peer to whom a gridlet has been submitted fails it is desirable a minimal loss of the already performed work. Therefore, the checkpointing mechanism shall be able to resume the execution from the last checkpoint. It is possible that the failed peer does not return, in that case it should be possible to resume the work in another peer. This is possible if the checkpoint file is saved outside the failing peer (e.g., a central server, the submitting peer, a set of peers).

GiGi applications are deterministic and do not communicate. Therefore, distributed checkpointing mechanisms and log mechanisms are not needed. Nevertheless, these mechanisms are described in the related work, for completeness.

[1] Validation - finding or testing the truth of something

[2] Verification - additional proof that something that was believed (fact, hypothesis or theory) is correct

2.3 Progress Monitoring

As an extension of the checkpointing system, we must be able to monitor the progress of the submitted gridlets. This can be useful for the user to know how much of the job has already been performed and may even enable the user to preview the results.

3 Related Work

This section presents the state of the art of this work's central topics. The next subsection describes peer-to-peer [33, 7, 5] and cycle-sharing [3, 21] technologies, section 3.2 describes how result verification is done in public computing systems and section 3.3 explains the checkpointing solutions [39, 15, 26] that are currently used.

3.1 Peer-to-Peer & Cycle Sharing

Peer-to-peer and Cycle-sharing systems have been merged enabling Public computing.

3.1.1 Peer-to-peer. Peer-to-peer lacks a consensual formal definition. Common users understand peer-to-peer as the genre of applications that allow them to be part of communities that cooperate by exchanging files. Their perception is biased by the most popular peer-to-peer systems deployed over the Internet so far. Projects like Napster [30] and KaZaA [20] have successfully enabled users to exchange files among themselves. More generally peer-to-peer can be defined as the genre of systems that take advantage of the resources (i.e., content, CPU cycles, storage space, human presence) located in the edges of a network (i.e., end user machines). These systems are composed by thousands of volatile participants. Peer-to-peer systems are capable of accommodating this transient population with minimal impact on the core business of the system (i.e., the exchange of resources). The sum of all the resources present in these systems often surpasses the resources owned by any institution. Therefore, the correct aggregation and use of these resources can unleash an enormous potential.

3.1.2 Peer-to-peer Applications. Peer-to-peer systems fall into one or more of following application categories: Distributed Computing, Content Sharing, Collaboration.

Distributed Computing Systems are characterized by taking advantage of computing power at the edges of the network to speed up the execution of CPU intensive tasks. In order to prevent the user's frustration due to a lack of computational power these applications may run as low priority processes or only when the computer is idle (acting like screen-savers [2]). All distributed computing systems work under the same assumptions: a computationally expensive task is divisible into smaller independent work-units; once these work-units have been executed, their results can be aggregated producing the result of the initial task. Obviously, only some computational tasks are appropriate to this model of execution and even if they are, they may not have a visible speed-up. Examples of these are Seti@Home [3], GIMPS [18], distributed.net [11], Folding@home [21], etc.

Content Sharing Systems are the most popular peer-to-peer systems. These systems appeared as means to circumvent the servers inability to provide large files to multiple users simultaneously, given their limited bandwidth. Considering a network composed by thousands of participants, a file can be replicated from dozens to hundreds of times depending on its popularity. This replication enabled users to download one file from several users, with no bandwidth bottlenecks. Distributed computing systems enable their participants to share their files within a community. Though the base service of some applications is legitimate, the participants have used them mostly to exchange copyrighted digital media. Some systems like Napster [30] have been shut down by the authorities for being considered as means for piracy. Though, Napster was an easy target due to its hybrid decentralized architecture, a pure peer-to-peer system is impossible to turn off effectively. Examples of these systems are Napster [30], KaZaA [20], Freenet [9], etc.

Collaboration Systems enable people to interact in real-time. The resource being shared is human presence. Since human presence is always located at the edges of the network, all instant messaging and multi-player gaming can be considered peer-to-peer. Instant messaging, audio/visual communication and on-line gaming are examples of collaboration systems.

3.1.3 Peer-to-peer Architectures. Considering the Client-Server Model, there exist two entities: the Client, and the Server. The client distinguishes itself from the server for always being the one that starts the communication. The client requests a resource to the server and the server replies with the required resource. A peer implements both the functionalities of client and server (other definitions for peer are node, or servent). A Peer-to-peer system is composed by a massive amount of interconnected symmetric peers. Nevertheless, some peer-to-peer systems require other entities than the peers to operate: super-peers or central servers. Considering this entities we can classify the peer-to-peer systems in terms of their network overlay centralization into: purely decentralized, partially centralized and hybrid decentralized architectures.

Purely Decentralized Architectures [40] are composed by peers. Every peer in the network implements the same functionality. Peers communicate directly with each other, the exchange of resources is done directly between two peers. This is peer-to-peer in its purest form: completely decentralized without single points of failure. However, searching can be a problem in these architectures.

Partially Centralized Architectures [20] are composed by peers and super-peers. Super-peers are peers whom have been assigned to perform additional tasks while maintaining their basic peer functionalities. They are chosen to become super-peers if they provide some abundant resource (e.g., bandwidth). The additional tasks are usually aggregation of knowledge about the system in order to improve performance (e.g., of searching). The exchange of resources is done directly between peers and super-peers. Since super-peers can be dynamically assigned, they do not constitute single points of failure or scalability issues (if a super-peer fails, another peer can be chosen to become a super-peer).

Hybrid Decentralized Architectures [30] are composed by peers and a central server. The direct exchange of resources is done between peers. The central server performs complementary, but essential, tasks (e.g., bootstrapping, indexing). Having a central server also means having a single point of failure and reduced scalability.

Beside these three degrees of centralization, there are other systems whom are considered by some authors as peer-to-peer systems. Seti@Home [3] is in the barrier that separates the client server model from the peer-to-peer model. In terms of architecture its approach resembles client-server since there is no communication or resource trading between peers. Conceptually, it takes advantages from resources at the edges of the Internet and therefore shall be considered peer-to-peer. D. Anderson has referred to the model as "inverted client-server", since the power resides on the edges of the Internet, the central server only coordinates it. Even though the "power" is decentralized, it does not fit in the more increased degree of decentralization defined for peer-to-peer systems, since there is no direct exchange of resources between the peers.

Table 1. Peer-to-peer systems architectural comparison

Model	Client-server	Inverted Client-Server	Peer-to-Peer		
Centralization	Centralized	Centralized	Hybrid Decentralized	Partially Centralized	Purely Decentralized
Entities	Clients and Server	Peers and Master Central Server	Peers and Central Server	Peers and Super-peers	Peers
Communication	Between client and server	Between peer and server	Between peer and server; between peers	between peers(Super-peers are also peers)	between peers
Resource Provider	Server	Peer	Peer	Peer	Peer
Resource User	Client	Server	Peer	Peer	Peer
Examples	Web Browsers and Web Servers	SETI@Home	Napster	KaZaA	Freenet, Ginger

Peer-to-peer systems can also be classified having into consideration the way their network overlay[3] is structured. It basically defines the connections that exist among the peers in a peer-to-peer system. Systems have been built with structured and unstructured network overlay topologies.

[3] The network overlay is the virtual network built on top of the real network

Unstructured Systems [20, 30] create their overlay in an ad-hoc manner, not following any specific rules. A peer is connected to a bunch of other peers at random. The placement of content and info is not related to the network overlay. Peer-to-peer is all about resources, being these resources scattered throughout the peers. We must be able to locate them. The usual searching mechanisms vary from flooding to other more elegant techniques (e.g., random walks, routing indexes). Nevertheless, these techniques have a limited scope which might become a problem when searching for a rare item, though they work well for popular content. Unstructured systems are generally more appropriate for accommodating highly-transient node populations, since the overhead incurred by a peer that joins or leaves the network is negligible. Napster [30], KaZaA [20] are examples of unstructured systems.

Structured Systems [34, 38] create an overlay that obeys to strict rules. A node knows and is known by a clearly defined set of other peers. In these systems, there is a clear mapping between the identifiers of a node/content and their location in the overlay. The mapping is done using hash functions, creating a distributed hash table indexing structure. This indexing allows nodes and contents to be located in the network within a few steps. Though, the accommodation of highly transient populations can generate a significant overhead, since the overlay has to be reorganized when a peer joins or leaves the system. Another disadvantage of these systems is their inability in locating content when an exact name or identifier cannot be provided. CAN [34] and Chord [38] are examples of structured systems.

Hybrid Systems combine the previous systems exploring the advantages of both, while avoiding their drawbacks. Building a system with two separate overlays (one structured and one unstructured) is possible. Though, this naive approach would generate high overheads. Pastry [35] and Kademlia [27] have developed more elegant techniques that conciliate structured and unstructured models.

3.1.4 Examples of Peer-to-peer Systems.

Napster [30] was one of the most popular peer-to-peer content distribution systems. In this system, all the indexing was stored in a central server. Peers queried the server for file locations, to further download from. The central server was an issue to scalability and constituted a single point of failure. It was shut-down by the authorities for it constituted a mean for piracy. Though, Napster became historical.

KaZaA [20] is an unstructured content distribution system. Searching is done through flooding methods. To reduce the network traffic overhead generated by the searching methods, peers with high bandwidth become super-peers that maintain an index of the files located in a set of peers with lower bandwidth.

Chord [38] was the first structured peer-to-peer system. Chord maps nodes and content using the same hash function, positioning them in a ring shaped overlay. Each node is responsible for maintaining a subset of the contents (or pointers to it), this subset is based in ranges of the hashes of the identifiers (e.g., between the node identifier and its successor identifier).

Content-Addressable Network (CAN) [34] is another structured peer-to-peer system. CAN places nodes and content in a virtual n-dimensional Cartesian space. Each node is responsible for a zone of the space.

Pastry [35] organizes the nodes in a circle according to their node identifiers, like Chord. It routes a message to the node whose identifier has the longest common prefix. In addition it maintains a table with the closest peers identifiers and locations.

Kademlia [27] assigns 160 bit identifiers to nodes and content using the SHA-1 function. The overlay can be seen as a binary tree. Every node knows at least one node in each of its sub-trees. This enables a node to find any other node in the network.

3.1.5 Cycle-sharing. Cycle sharing systems are distributed systems that execute a long running application in parallel in order to speed it up. These high performance computing systems are composed by non-dedicated machines. This systems emerged in institutional environments [23] and were later transposed to public environments [3]. Some problems that were already addressed in the institutional environments had to be reconfigured to public environments (e.g., resource location). Other issues, like fairness, trust, incentives and security, are current research topics.

3.1.6 Cycle-sharing Architectures. The majority of the cycle-sharing systems developed are based on an inverted client-server architecture. In these architectures, there is no communication between clients. All the clients communicate with a central server only. These architectures are fit for projects that harness idle cycles of hardware owned by the general public to perform computation related with global causes. Figure 1 shows an inverted client-server architecture where a server communicates with a group of heterogeneous machines.

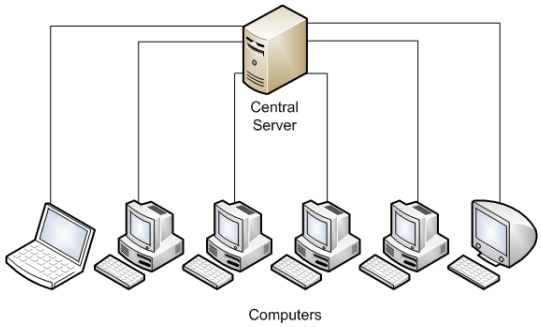

Fig. 1. Inverted client-server architecture in cycle-sharing systems.

Cluster Computing On the Fly [25] proposed a complex architecture where users join communities depending on how they would like to donate their cycles. These communities are transformed into community-based overlay networks. Then, clients form

a computer cluster on the fly from these overlays. Figure 2 shows the architecture of Cluster Computing On the Fly.

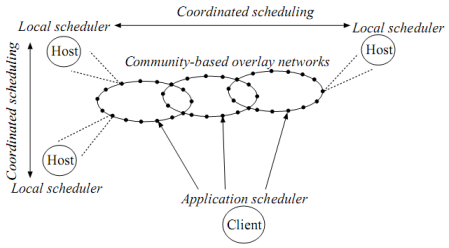

Fig. 2. Architecture of Cluster Computing On the Fly.

3.1.7 Examples of Cycle-sharing Systems.

Seti@Home [3] is the most successful public computing project. With the objective of finding extraterrestrial life, it analyses radio signal. This analysis requires a considerable amount of processing power. This project was not able to afford hardware to execute this task. As a solution, they developed a client that the public could download. The client fetches work from the server and returns results to it. Motivated by rankings and communities, the number of participants grew much more than they expected. Nowadays, they have more processing power than what they need.

BOINC [2] is an infrastructure that allows the development of public computing projects. Nowadays, Seti@Home uses BOINC. By downloading the BOINC client, the general public can offer their idle cycles to several projects (e.g., climateprediction.net, Einstein@Home, Spinhenge@home, etc.). Though, to get a new public computing project working it is necessary to manage a complex server configuration.

Condor [23] organizes a group of machines into a cluster. This cluster is used like a pool of resources. If a user has a long running task to accomplish it schedules the work to the idle machines in the pool.

Cluster Computing On the Fly [25] empowers the common users to speed-up their computations by creating a cluster of computers on the fly.

distributed.net [11] harvests cycles through its client. It uses that processing power to crack encryption algorithms in a brute force manner (testing every possible combination). It has cracked RC5-56 bit and RC5-64 bit and is currently working on RC5-72 bit. This project has monetary prizes to attract its users.

Great Internet Mersenne Prime Search (GIMPS) [18] searches for Mersenne prime numbers (i.e., prime numbers that can be written in the format: $2^p - 1$). It also has monetary prizes to attract users. Participants run a client, that executes and returns work to the server.

Folding@Home [21] is Stanford University project that studies how proteins fold. This may reveal cure for yet non-curable diseases. It has accomplished several folding simulations since 2001. To participate, users have to install a client that communicates with the server.

3.2 Reliable Result Verification and Validation

To speed up the execution of long running algorithms, high performance computing systems schedule jobs to be executed in other places. Confirming results of third party executions is a hard task. An exhaustive verification of these results would cause a major slow down in the system, contradicting its objective.

3.2.1 Bad Results. Wrong results have different origins and objectives. They can be distinguished into faulty results, malicious results and cheating results.

Faulty Results have no objective, they are originated by faults or byzantine behaviour. Both usually produce incoherent results and are relatively easy to identify, when compared with results created with malicious motivations.

Single Malicious Results intend to harm the system. They are attacks that explore the vulnerabilities of the system, causing it to work inappropriately. One job's bad result corrupts the whole long running application result.

Collective Malicious Results intend to harm systems that use replication as a mean to verify their results. To perform these attacks several participants return the same bad result [12]. This is know as collusion.

Cheating Results are returned by cheating participants to receive credit for the work they have not performed. The majority of the cycle sharing systems values ranking tables, and some participants are willing to corrupt their results to go up in those rankings [28].

3.2.2 Techniques for Identifying Bad Results. Several techniques have been proposed to identify bad results. Techniques vary in their complexity, overhead and effectiveness. Though, no single technique can identify all the four types of bad results we have defined. However, it has been demonstrated that these techniques combined with a reputation system can improve the reliability of the results produced by the system [41].

Replication [3] is one of the most effective methods to identify bad results with redundant execution and comparison between results. In these schemes the same job is performed by *N* different participants (*N* being the replication factor). The results are compared using voting quorums, and if there is a majority the corresponding result is accepted. Since, it is virtually impossible for a fault or a byzantine behaviour to produce the same bad result more than once, this technique easily identifies and discards the bad ones. However, if a group of participants colludes it may be impossible to detect a bad result. Another disadvantage of redundant execution is the overhead it generates, since every job is executed at the very least two times. Most of the public computing projects use replication to verify their results, it is a high price they are willing to pay to ensure their results are reliable. Seti@Home has a fixed amount of information to be processed and since they have more computing power than they need, replication is far from being a problem to them. Projects like Folding@Home have to choose which simulations should be performed first and which ones will have to wait.

Replication consumes at the very least twice more resources than the ones that are actually needed to perform the execution in order to produce more believable results. When there is no collusion, it is virtually capable of identifying all the bad results with 100% certainty.

Hash-Trees [13] defeat cheating participants by forcing them to calculate a binary hash-tree from their results, and return it with them. The submitting peer only has to execute a small portion of a job and calculate its hash. Then, when receiving results, the submitting peer compares the hashes and verifies the integrity of the hash-tree. This dissuades cheating participants because finding the correct hash-tree requires more computation than actually performing the required computation and producing the correct results. Figure 3 shows a hash tree where the leafs are partitioned sequential results or the data we want to check. The hash is calculated using two consecutive parts of the result concatenated, starting by the leafs. Once the tree is complete, the submitting peer executes at random a small portion of the whole work (the selected sample) that corresponds to a leaf. Then this result is compared to the returned result and the hashes of the whole tree are checked.

Hash-trees make cheating unworthy. They have a relative low overhead: a small portion of the work has to be executed locally and the hash tree must be checked.

Quizzes [25] are jobs to whom the result is known by the submitter a priori. Therefore, it can test the honesty of a participant. Cluster Computing On the Fly proposed two types of quizzes: stand-alone and embedded quizzes.

Stand-alone quizzes are quizzes disguised as normal jobs. They can test if the executing node executed the job. These quizzes are only useful when associated with a reputation system that manages the trust levels of the executing peers. Though, the use of the same quiz more than once can enable malicious peers to identify the quizzes and to fool the reputation mechanisms. The generation of infinite quizzes with known results incurs considerable overhead.

Embedded quizzes are smaller quizzes that are placed hidden into a job, the job result is accepted if the results of the embedded-quizzes match the previously known ones. Embedded quizzes can be used with or without a reputation system. Though, their

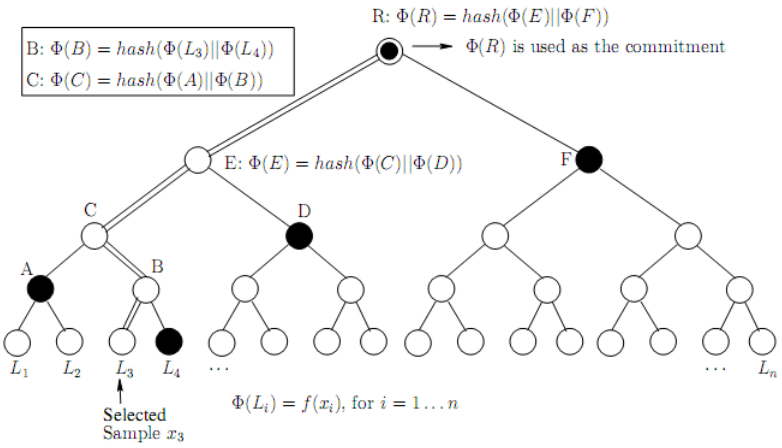

Fig. 3. Example of an hash tree.

implementation tends to be complex in most cases. Developing a generic quiz embedder is a software engineering problem that has not been solved so far.

3.2.3 Reputation Mechanisms. Reputation mechanisms [19, 41] are inherent to security and not fault-tolerance. So, they are considered out of the scope of this work. Though, a reputation manager can be built between the result verifier and the scheduler, being fed by the result verifier it is able to manage the scheduler (e.g., blacklisting, ranking).

3.3 Checkpoint Restart

Checkpointing is a primordial fault tolerance technique. Long running applications usually implement checkpointing mechanisms to minimize the loss of already performed work when a fault occurs. Checkpoint consists in saving a program's state to stable storage during fault-free execution. Restart is the ability to resume a program that was previously checkpointed. In high performance computing systems checkpoint/restart mechanisms are not only used for fault mitigation, they enable these systems to migrate the jobs taking the best advantage of the systems present resources (i.e., load balance). Migration [14] is the resuming of an application that was checkpointed elsewhere (on another machine).

3.3.1 Implementation Approach. Checkpointing mechanisms can be distinguished by their implementation approach into Application-level, Library-level and System-level.

Application-level Checkpointing Systems [3, 21] do not use any operating system support. These are usually more efficient and produce smaller checkpoints. They also have

the advantage of being portable[4]. These checkpointing mechanisms are implemented within the application code. They require a big programming effort. Checkpointing support built in the application is the most efficient, because the programmer knows exactly what must be saved by to enable the application to restart in case of failure. Though, this approach has some drawbacks: it requires major modifications to application's source code (its implementation is not transparent to the application); the application will take checkpoints by itself and there is no way to order the application to checkpoint if needed; it may be hard, if not impossible, to restart an application that was not initially designed to support checkpointing; and it is a very exhaustive task to the programmer. This programming effort can be minimized using pre-processors that add checkpointing code to the application's code, though they usually required the programmer to state what needs to be saved (e.g., through flagged/annotated code). Though, not all applications are fit for this approach.

Library-level Checkpointing Systems [31, 24] consist in linking a library with the application, creating a layer between the application and the operating system. This layer has no semantic knowledge of the application and cannot access kernel's data structures (e.g., file descriptors), so this layer has to emulate operating system calls. The major advantage is that a portable generic checkpointing mechanism could be created, though it is very hard to implement a generic model to checkpoint any application. This checkpointing method requires none or very few modifications to the applications source code.

System-level Checkpointing Systems [42] are built as an extension of the operating system's kernel, therefore they are more powerful. They can access kernel's data structures. Checkpointing can consist in flushing all the process's data and control structures to stable storage. Since these mechanisms are external to the application they do not require specific knowledge of the application, and they require none or minimal changes to the application. They have the obvious disadvantage of not being portable and usually more inefficient than application-level.

3.3.2 Checkpointing Distributed Applications. Various techniques have been proposed that enable the checkpointing of distributed applications. These techniques can be divided into coordinated checkpointing, uncoordinated checkpointing and message-induced checkpointing.

Uncoordinated Checkpointing [15] tries to find a match between the checkpoints taken by each of the processes to create a global checkpoint. Each of the processes can take checkpoints independently. This is an advantage because not all applications can checkpoint at any time. Still, this technique has some drawbacks: there are chances of occurring as domino effect[5] [6]; it may create checkpoints that are useless, as they are never chosen to be part of a global state; and multiple checkpoints must be kept in storage, in order to choose the one that fits the global checkpoint.

[4] Portability is the ability of moving the checkpoint system from one platform to another

[5] Domino effect is the impossibility of creating a global checkpoint from the local checkpoints taken, which may cause the application to restart from the beginning

Coordinated Checkpointing [15, 8] makes processes cooperate to create a global consistent checkpoint. This reduces the storage space required to save checkpoints, since only one checkpoint is persisted at a given time. The major disadvantage of this approach is the amount of communication required to perform a global consistent checkpoint, causing this technique to have scalability problems. The algorithms to perform coordinated checkpointing vary in its complexity. Easy to implement techniques have high communication overhead, more complex techniques have been proposed to minimize this overhead, such as: non-blocking checkpoint coordination, synchronized checkpoint clocks, and minimal checkpoint coordination.

Communication-induced Checkpointing [15] combines coordinated and uncoordinated checkpointing methods in order to avoid the domino effect and allow processes to checkpoint more autonomously. It works by appending checkpointing information to the application messages (piggy-backing). This checkpoint information is used to determine whether the process must take a checkpoint or not. This method does not require special coordination messages to be exchanged between the processes, lowering the communication overhead.

3.3.3 Non-determinism Support. To be able to checkpoint non-deterministic applications the checkpointing system must implement logging mechanisms. Non-deterministic events, such as the receipt of a message or user input, must be recorded to the log, so they can be replayed later if needed.

Pessimistic Message Logging [15] logs each event to stable storage before delivering it to the application, assuming that a fault may occur between the event and the logging of that event. The advantages are simplified restart mechanisms and ease to identify the logged events that can be discarded once a checkpoint has taken place. However, this produces high overheads.

Optimistic Message Logging [15] logs the event to volatile storage instead of stable storage. The events are periodically flushed from memory to disk. This greatly reduces the overheads, however restart mechanisms are complex and events can be lost.

Causal Message Logging [15] tries to take advantage of the previous methods. Events are stored to volatile storage, but are replicated to other processes or applications. It also periodically flushes these events to stable storage. This method has a better performance than the pessimistic and avoid the loss of events of the optimistic. However, events may be lost due to the failure of several processes or applications.

3.3.4 Checkpointing Enhancements. Checkpointing systems always incur overhead during fault-free execution. The major source of overhead is the stable storage access. In order to reduce this overhead some enhancements have been proposed.

Concurrent Checkpointing [15] aims at reducing the time a process is blocked due to a checkpoint operation. While the process is being checkpointed it remains blocked so he cannot modify its memory. Concurrent checkpointing reduces the time that a process remains blocked by marking its memory copy-on-write. This allows the process to be unblocked during the checkpointing.

Incremental Checkpointing [16, 22] avoids rewriting portions of the process state that did not change between checkpoints. Minimizing the amount of data to be written lowers the time required to store the checkpoint. After the creation of a checkpoint, state changes are logged incrementally. It is possible to lower the time interval between checkpoints or in the extreme not to use it (i.e., propagate the program state changes to the checkpoint as they occur). This has a constant, but small, overhead. At any time the checkpoint represents the current state of the application. There is no need for complex algorithms for estimation of the perfect checkpointing interval, and none of the already performed work is ever lost.

Diskless Checkpointing [32] uses volatile memory to store checkpoints which provides decreased storage times. This can be done using the same machine's memory or using other machine's memory. However, the checkpointing data can be lost due to the failure of a computer. To address this problem, the checkpointing data is periodically copied to persistent storage or sent over the network to others (replication).

3.3.5 Examples Checkpoint Restart Systems.

Libckpt [31] is a virtually transparent checkpointing mechanism (there is a minimal amount of the application's code that has to be modified). It provides a user-level library that can be linked with user's applications, providing them with checkpointing mechanisms. It is not portable, it has been designed to execute on UNIX. Libckpt is very limited because it cannot access system states maintained by the kernel.

CRAK [42] is a Linux kernel module that implements checkpoint/restart mechanisms. It requires no modifications to the user's applications, but requires modifications in the operating system. Therefore, it is transparent but not portable. It has access to all kernel states needed to checkpoint an application correctly. Though it is one of the most complete checkpointing systems, it is far from supporting any application.

4 Architecture

In this section it is described what is expected to be implemented. First, we present an overall of the architecture of GiGi. Then, we address techniques of result verification and checkpointing that will be used. And finally, a draft of the fault-tolerant architecture to be developed is explained.

4.1 Architecture of GiGi

GiGi proposed a cycle-sharing system where every peer is able to submit and execute jobs. GiGi is only attractive if the users are able to run a vast range of popular applications. Therefore, the system must execute the applications unmodified. To achieve this, GiGi proposed the concept of a gridlet. A gridlet is composed by the data and the code (or a pointer to it) to be executed over the data. Figure 4 shows an overall of the GiGi architecture.

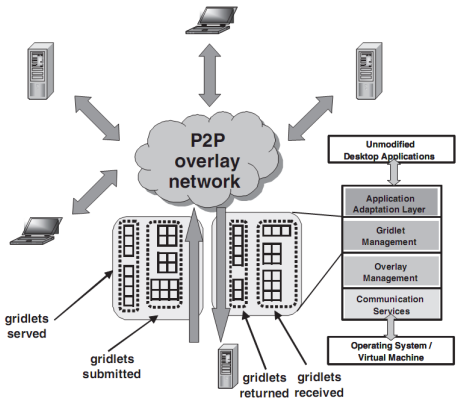

Fig. 4. Architecture of GiGi.

4.2 Result Verification and Validation Techniques

As seen in section 3.2 replication is the most effective technique for result verification and validation. To verify the results we will implement replication with different flavours and inverted quizzes.

4.2.1 Replication with Standard Partitioning. High performance systems partition a long running task into smaller jobs. Each of the jobs is executed redundantly at different participants. With this, the same job is executed at least twice. This technique is vulnerable to colluding participants. They can easily identify the same job is being performed in more than one of the members of the malicious group and agree to return the same bad result.

4.2.2 Replication with Overlapped Partitioning. In this form of replication, the jobs to be executed are never equal. With this it becomes more complex for the colluders in the group to identify the work that they are doing redundantly. Plus, they will have to perform at least a part of the job.

In order to create different jobs, one long running task has to be partitioned N times (N being the replication factor) with variable parameters, preferably random. This is always possible, since the breakdown of a long running is possible to do in different ways. Figure 5 shows a draft of the same task broken in two different ways. What is being done is creating more points of comparison.

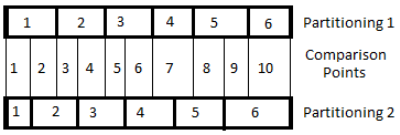

Fig. 5. Overlapped partitioning with replication factor 2.

Overlapped partitionings can also enable a more relaxed replication technique in which not all the work is redundantly executed. Figure 6 depicts a technique of partitioning for relaxed replication. This lowers the overhead of replication while maintaining some points of comparison that still enable us to identify bad results.

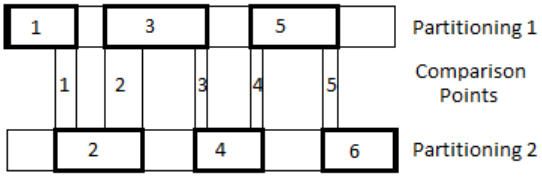

Fig. 6. Overlapped partitioning for relaxed replication.

4.2.3 Replication with Meshed Partitioning. Meshed partitionings can create even more comparison points, the overlapped partitioning described above is always implementable. But for some specific kinds of applications an n-dimensional partitioning is possible. For example, for a ray tracer we can divide an image into 10 sets of lines in the first partitioning, and into 10 sets of columns in the second partitioning. For replication factor 2, overlapped partitioning produces 10 comparisons, while meshed partitioning produces 100 comparisons. Figure 7 shows a meshed partitioning example for images.

The replication factor is usually odd for ease of determination of majorities. Though, even replication factors can save effort in the most cases (when there are no errors nor attacks). In case of a draw a job can be assigned for re-execution on a different peer.

We are also working in a stateless reputation mechanism (using meshed partitionings). It would provide means to decide between two different results. This would work by calculating a ratio using the results of the comparison points. The ratio would be the

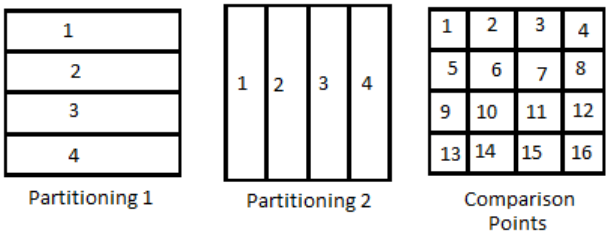

Fig. 7. Meshed partitioning for images using replication factor 2.

number of equal comparison points divided by the total number of comparison points of a work unit. When there is no match in a comparison point, the result of the work-unit with higher ratio would be chosen. Though, this can create some tricky cases and will needs further research.

4.2.4 Inverted Quizzes. Quizzes are small jobs with known results that are executed by the participants, these can be divided into stand-alone and embedded quizzes (as described in section 3.2). Though, quizzes have some drawbacks: they have to be generated; one quiz must be used only once; they have to be hidden inside a task (this may be hard to implement).

Inverted quizzes consider the execution of the smallest unit of execution possible (a sample) in the submitting peer. One sample is chosen randomly for each work unit. Since this sample is executed locally it is a trusted result. Once the sample is compared with the corresponding fraction of the result of a work unit, that result can be accepted or discarded. Figure 8 shows the partitioning of an image into four work units (each work unit is a set of lines of the whole image). The samples are randomly chosen pixels, one for each work unit.

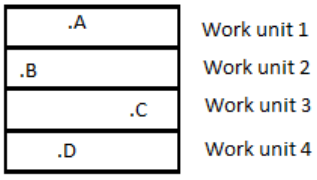

Fig. 8. Example of sampling for images using inverted quizzes.

4.2.5 Summary. GiGi will have three levels of verification. Firstly, the results will be validated in terms of file format and comparison with previous results, this will discard some result with very low effort; secondly, with replication we will be able to choose

the result through voting quorums; and thirdly, the result will be checked using inverted quizzes. Only if a result passes all these tests it will be accepted. Along with the results, the claims of costs will also be verified. This will be done by detecting large variations in these claims.

We will also enable the user to establish the degree of trust he is willing to pay for the reliability of his results. This will be done by varying the parameters of the previous techniques (e.g., the replication factor, the number of quizzes/samples per work-unit).

4.3 Checkpointing

At least two techniques will be implemented. Any of the techniques must be portable and transparent to the applications. One will be a generic technique that will fit any application and the other will be a result directed technique, that may be more efficient for some applications.

4.3.1 Generic Checkpointing. This checkpointing mechanism will be a library-level or a virtual machine with freezing and unfreezing capabilities.

Library level implementations do not have full support over everything that is needed to perform checkpointing of a vast range of applications. Though, they might be fit for some applications.

Virtual Machine implementation has some desirable properties, other than checkpointing, like security of the executing peer. Minimizing the risk of executing code provided by malicious submitting peers. For matters of checkpointing, the image of a frozen virtual machine is huge, this creates great overhead in the network. This approach can be enhanced using a technique called Just enough Operating System (JeOS or "juice"). This consists of a minimized operating system that provides only the support needed by a particular application. As another enhancement, we can checkpoint only the differences between the current state and the start-point. The start-point is a checkpoint created before the launch of any application. The start point must exist in any peer (or be accessible). Using start-points lowers the overhead on the network. To start a new job, the start-point is resumed and the application is invoked. To resume a checkpoint, the differences are propagated to the start-point and then it is resumed.

4.3.2 Result Checkpointing. This technique of checkpointing arises from the genre of applications that Ginger is fit for. These kinds of applications usually produce final results and write them to a file (e.g., an image file). These files can be a form of checkpoint files if we are able to resume the execution from them. For example, a ray-tracer has written an image with 13 lines before failing (the image file is the checkpoint file), then if we are able to resume the ray-tracer in line 13 this is a checkpoint restart mechanism. These result files can be monitored, or the file system calls can be intercepted. Though, some assumptions are being made: the application has mechanisms that allow the resuming somehow (at least invoking them with different parameters or input files). However, for the applications, that we are able to do this, the checkpointing mechanisms

will be very efficient. These chekpointing files are the result files, and if they are sent to the submitting peer incrementally, no overhead size is added by the checkpointing mechanisms. The execution overhead will also be very low.

4.4 Proposed Fault Tolerant Architecture

Figure 9 depicts the architecture to be developed. It show the relevant modules and their interactions. Each module has a specific function. Not all the modules present in this architecture are for fault tolerance, some of them will only be bots (i.e., they will simulate behaviour). The bot modules are the reputation manager and the resource manager. Next each of the modules is briefly described.

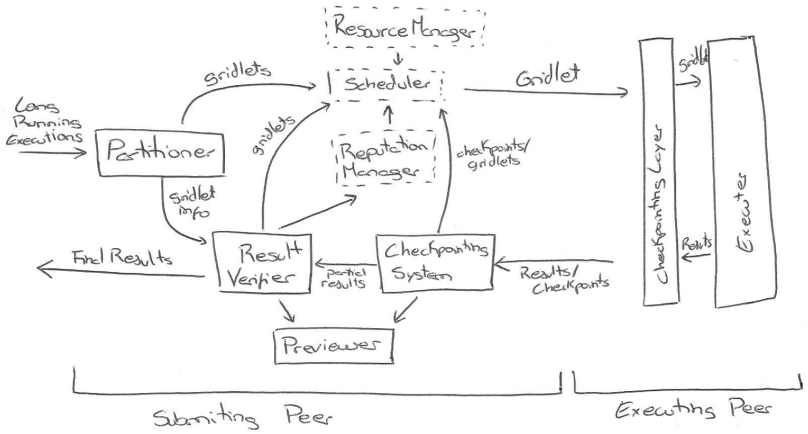

Fig. 9. A draft of the GiGi fault tolerant architecture.

The Partitioner breaks a long running task into small jobs (i.e., gridlets), and sends them to the scheduler. It also provides partitioning info to the result verifier.

The Scheduler decides which participants will execute which gridlets. It bases its decision in the information provided by the resource manager and the reputation manager. It receives gridlets from the partitioner, the checkpointing system and the result verifier.

The Reputation Manager receives information about the correctness of the results received from the participants. This information is analysed in order to influence the Scheduler's choices.

The Result Verifier compares the received results and checks their correctness. It is also responsible for aggregating the results; providing information to the Reputation Manager; and actualize the previewer with final partial results. If needed, it can submit gridlets to be re-executed.

The Checkpointer manages checkpoints and detects failures of participants. It feeds the Result Verifier and the Previewer with partial results. The previewer can show partial results enclosed in the checkpoints. Upon participant failure detection, it must be capable to build a gridlet that consists on the resumption of a task from a previous checkpoint and submit it to the scheduler

The Previewer lets the user preview the received results (verified and unverified). Preview is better for results like images, for less visual application it can display percentages of progress and other interesting data.

5 Evaluation

The fault-tolerant mechanisms to be developed will further be evaluated. Result Verification mechanisms and Checkpointing mechanisms will be evaluated separately. In order to be able to compare the new techniques proposed, standard techniques will also be implemented. The next subsection describes what will be possible to prove in the evaluation process.

5.1 Evaluation of the Result Verification Mechanisms.

These mechanisms will be evaluated having into consideration the various types of faults that have been defined in section 3.2. Faulty, single malicious and cheating results generate two types of results: totally erroneous and localized errors. Malicious collectives produce the same bad result, totally erroneous and localized results. So two different scenarios will be emulated. Though the result verification may not be completely reliable (it may eventually accept wrong results, e.g., a valid yet wrong result provided by a majority of still reputable colluding nodes that nonetheless solved quizzes), we must ensure that no good results are discarded. The evaluation measurements will focus on the percentages of bad results accepted.

5.2 Evaluation of the Checkpointing Mechanisms

Checkpointing mechanisms will be evaluated considering their correctness and their overhead.

Correctness. Correctness is proved if the checkpointing mechanism is able to checkpoint and restart jobs no matter at what point the fault occurs. To prove this, faults will be injected randomly and at specific sensible points of execution. This means correctness will be proven in a faulty environment.

Overhead. The major disadvantage of all the checkpointing systems and techniques is their overhead. The overhead will be measured in a fault-free environment. Two different types of overhead will be measured: the execution time and the checkpoint size. The checkpoint size is very important because this checkpoints are to be sent through the

network. So size is directly related to bandwidth overhead. At least two scenarios will be executed, and the conclusions will stem from the comparison of the obtained values. The scenarios are:

Base-line tests will measure the time to perform a given execution in a fault-free environment with the checkpointing mechanisms disabled. This will give the base execution times. The checkpoint size is always zero in this scenario, obviously.

Checkpointing tests will be executed with the checkpointing mechanisms enabled. In order to compare the different checkpointing mechanisms implemented, several checkpoints will be tested. The difference between the time of execution of these tests and the base tests will give the time execution overhead. The size of the checkpoints generated during the execution will also be recorded. The size will vary during the executions, considering the creation of several checkpoints per execution. The size of the checkpoint will be the mean of the generated checkpoints. The time execution overhead and the size of the checkpoints will be compared between the different checkpointing methods.

6 Conclusions

We have shown that cycle-sharing systems have new challenging issues. Fault tolerance mechanisms can improve the reliability of these systems. In this work we explored how can we improve the reliability of the results and how can we ensure progress in execution in a faulty environment.

For result verification, we have proposed replication with new flavours (overlapped, meshed and relaxed) and inverted quizzes. Using these techniques with variable parameters, we will enable the user to define how much he is willing to pay for the reliability of his results.

To ensure that the progress of a work unit is not completely lost when a peer fails, we have proposed two checkpointing techniques: result checkpointing and generic checkpointing. Result checkpointing, is based in restart mechanisms that use result files as checkpoint files (this will be efficient but not fit for all applications). Generic checkpointing will ensure that we are able to checkpoint any application.

Bibliography

[1] D. P. Anderson. Public computing: Reconnecting people to science. 2002.

[2] D. P. Anderson. Boinc: a system for public-resource computing and storage. pages 4–10, 2004.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations, 1995.

[5] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[6] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. On modeling consistent checkpoints and the domino effect in distributed systems, 1995.

[7] D. Barkai. Technologies for sharing and collaborating on the net. *Peer-to-Peer Computing, IEEE International Conference on*, 0:0013, 2001.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–??, 2001.

[10] L. B. Costa, L. Feitosa, E. Araujo, G. Mendes, R. Coelho, W. Cirne, and D. Fireman. Mygrid: A complete solution for running bag-of-tasks applications. In *In Proc. of the SBRC 2004, Salao de Ferramentas, 22nd Brazilian Symposium on Computer Networks, III Special Tools Session*, 2004.

[11] distributed.net. In *http://distributed.net/*.

[12] J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[13] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 4–11, 2004.

[14] J. C. e Alexandre Sztajnberg. Introdução de um mecanismo de checkpointing e migração em uma infra-estrutura para aplicações distribuídas. 2008.

[15] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems, 1999.

[16] T. H. Feng and E. A. Lee. Incremental checkpointing with application to distributed discrete event simulation. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1004–1011. Winter Simulation Conference, 2006.

[17] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[18] GIMPS. Great internet mersenne prime search. In *http://mersenne.org/*.

[19] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM.

[20] Kazaa. In *http://www.kazaa.com/*.

[21] S. M. Larson, C. D. Snow, M. Shirts, V. S. P, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology.

[22] J. L. Lawall and G. Muller. Efficient incremental checkpointing of java programs. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (for-*

merly FTCS-30 and DCCA-8), pages 61–70, Washington, DC, USA, 2000. IEEE Computer Society.

[23] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[24] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[25] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In *In Proceedings of the IEEE Fourth International Conference on Peer-to-Peer Systems*, pages 227–236, 2004.

[26] A. Maloney and A. Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurr. Comput. : Pract. Exper.*, 21(12):1632–1666, 2009.

[27] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. pages 53–65, 2002.

[28] D. Molnar. The seti@home problem, 2000.

[29] M. W. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Performance '87: Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 529–544, 1988.

[30] Napster. In *http://free.napster.com/*.

[31] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[32] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[33] R. Ranjan, A. Harwood, and R. Buyya. Peer-to-peer-based resource discovery in global grids: a tutorial. *Communications Surveys & Tutorials, IEEE*, 10(2):6–33, 2008.

[34] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[35] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[36] C. Shirky. Clay shirky's writings about the internet. In *http://www.shirky.com/*.

[37] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[39] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ACM Computing Research Repository (CoRR*, 501002:1–11, 2005.

[40] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses", December 2006.

[41] S. Zhao, V. Lo, and C. GauthierDickey. Result verification and trust-based scheduling in peer-to-peer grids. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 31–38, 2005.

[42] H. Zhong and J. Nieh. Crak: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science. Columbia University., November 2002.