INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Mecanismos para a Fiabilidade (Replicação, Tolerância a Faltas, Checkpointing) de Execução de Tarefas em Ambientes Cycle-Sharing

João Filipe Ramos Paulino

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor Alberto Manuel Ramos da Cunha
Orientador:	Professor Doutor Luís Manuel Antunes Veiga
Co-orientador:	Professor Doutor Paulo Jorge Pires Ferreira
Vogais:	Professor Doutor David Manuel Martins de Matos

Outubro 2010

Resumo

O particionamento de tarefas de longa duração em pequenas tarefas que são executadas paralelamente em varias máquinas pode acelerar a execução global da tarefa de longa duração. Esta técnica foi explorada em *Clusters*, em *Grids* e mais recentemente em sistemas *Peer-to-peer*. Contudo, a transposição destas ideias de ambientes controlados (e.g., *Clusters* e *Grids*) para ambientes públicos (e.g., *Peer-to-peer*) levanta alguns desafios quanto à sua fiabilidade: será que um participante vai devolver o resultado da tarefa que lhe foi adjudicada ou vai falhar durante a sua execução? E se devolver um resultado, será esse o verdadeiro resultado da tarefa ou serão apenas dados aleatórios? Estes desafios exigem a introdução de mecanismos de verificação de resultados e de *checkpoint/restart* que melhorem a fiabilidade dos sistemas de computação de alto desempenho que envolvam a partilha de ciclos em ambientes públicos. Neste trabalho, propomos e analisamos uma abordagem em duas vertentes: i) mecanismos de *checkpoint/restart* que permitem mitigar a natureza volátil dos participantes; e ii) estratégias de verificação de resultados que permitem aferir a correcção dos mesmos.

Abstract

The partitioning of a long running task into smaller tasks that are executed in parallel in several machines can speed up the execution of a computationally expensive task. This has been explored in Clusters, in Grids and lately in Peer-to-peer systems. However, transposing these ideas from controlled environments (e.g., Clusters and Grids) to public environments (e.g., Peer-to-peer) raises some reliability challenges: will a peer ever return the result of the task that was submitted to it or will it crash? and, even if a result is returned, will it be the accurate result of the task or just some random bytes? These challenges demand the introduction of result verification and checkpoint/restart mechanisms to improve the reliability of high-performance computing systems in public environments. In this paper, we propose and analyse a twofold approach: i) checkpoint/restart mechanisms to mitigate the volatile nature of the participants; and ii) result verification strategies to improve the reliability of the results.

Palavras Chave
Keywords

Palavras Chave

Tolerância a Faltas

Fiabilidade

Verificação de Resultados

Checkpoint/Restart

Partilha de Ciclos

Computação Pública

Keywords

Fault-tolerance

Reliability

Result Verification

Checkpoint/Restart

Cycle-sharing

Public Computing

Index

List of Figures

List of Tables

1 Introduction

The execution of long running applications has always been a challenge. 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, some cannot afford them. The idea of executing this in several common machines in parallel was first explored in networks or clusters of workstations (Sterling et al. 1995; Anderson et al. 1995), 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 (Anderson et al. 2002; Larson et al. 2009), 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 have 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 (Mutka & Livny 1988) 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 (Foster & Kesselman 1996) and MyGrid (Costa et al. 2004) have studied the imple-

mentation of grid infrastructures that provide high-level meta-computing services enabling the efficient development of applications. The Condor project (Litzkow et al. 1988) 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 (Shirky 2002) 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 (distributed.net 1997) and GIMPS (GIMPS 2010).

Public computing projects, so far, focus on mankind-related causes: Seti@home (Anderson et al. 2002) analyses radio signals trying to find evidence of extraterrestrial life; Folding@home (Larson et al. 2009) searches for the cures of diseases like cancer, Parkinson's, Alzheimer's, etc. studying how proteins fold; distributed.net (distributed.net 1997) solves brute-force cryptographic challenges exposing vulnerabilities; GIMPS (GIMPS 2010) 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 (Anderson 2003) 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)

GINGER (Veiga et al. 2007) proposed a network of favours where every peer is able to submit its work-units to be executed on other peers and execute work-units submitted by other peers as well. GINGER combines institutional grid infrastructures, distributed cycle sharing and decentralized Peer-to-peer architectures. GINGER 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). "* (Anderson 2003)

In order to be able to run an interesting variety of applications, GINGER 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). GINGER 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).

1.5 Objectives and Contributions

The migration of distributed computing systems from controlled environments (e.g., Clusters, Grids) to public environments (e.g., Peer-to-peer) raises several challenges. While Clusters and Grids make use of machines that 1) are managed by IT professionals, 2) have uptimes of 24 hours per day and 3) are trustful, Peer-to-peer faces issues of malicious behaviour, highly transient participants and users without enough IT know-how.

This work analyses mechanisms that improve the robustness of public computing systems: result verification strategies to determine the correctness of the returned results; and checkpoint/restart mechanisms to minimize the negative impact of volatile participants.

1.5.1 Result Verification

The untrusted nature of the participants demands the verification of every partial result, the returned results may be wrong due to the occurrence of a fault or malicious behaviour. The system must be able to identify these bad results, discard them and ensure that a correct one will be computed and accepted.

The techniques used to identify bad results incur considerable overhead. None of the result verification techniques developed is able to ensure with 100% certainty that a result is correct, though in some cases they can identify an incorrect one. The degree of certainty that a result is correct usually grows along with the overhead the technique incurs. Therefore, a compromise between the overhead and the reliability of the results can be found, this compromise must be dynamically adaptable to the variable conditions/resources of the system.

1.5.2 Checkpoint/Restart

The volatile nature of the participants can have a negative impact on the performance of the execution. Once a task has been assigned to a participant, we have no guaranties whatsoever that: 1) the participant is executing the task; 2) the participant will not fail/leave during the execution. Upon detection of one of these, the system reschedules the task to another participant and waits for the results to be retrieved. The rescheduling may happen several times, or possibly infinite times if no participant is available enough time to complete the task, leading to the never ending of a task.

To overcome this issue, the majority of the 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 from the point when it was saved. This minimizes the loss of the already performed work on the occurrence of a fault, allowing the rescheduling of the remaining work only.

Providing a specific application with checkpoint/restart capabilities is always possible if the application is being developed from scratch. However, in GINGER we face the challenge of providing a wide range of already existing applications with these capabilities without modifying them. Checkpointing systems may also enable the monitoring of the execution, the previewing of the results and the work migration (for matters of performance).

1.6 Document Structure

The next Chapter analyses the state of the art in the following areas: peer-to-peer, cycle-sharing, result verification and checkpoint/restart. Chapter 3 describes: the GINGER overall architecture; the faults and non-regular behaviour that must be handled; several result verification strategies based in replication and sampling; and two checkpoint/restart enabling techniques, through a virtual machine's running image, and through the result files. Chapter 4 describes our twofold implementation: a simulator that returns metrics that are used to evaluate the behaviour of the result verification strategies in environments with large populations; and a real deployment that proves that our techniques are easily implementable. Chapter 5 makes an evaluation of the proposed result verification and checkpoint/restart techniques considering

the benefits they provide against the overhead they incur. Chapter 6 concludes.

1.7 Scientific Publications

A preliminary version of this work was partially described in a scientific paper published and presented at INForum 2010, under the title *Exploring Fault-tolerance and Reliability in a Peer-to-peer Cycle-sharing Infrastructure*, and can be found at http://inforum.org.pt/INForum2010/papers/computacao-distribuida-de-larga-escala/Paper106.pdf.

2 Related Work

This Chapter presents the state of the art of this work's central topics. The next Section analyses peer-to-peer systems (Ranjan et al. 2008; Barkai 2001; Androutsellis-Theotokis & Spinellis 2004). Section 2.2 describes cycle-sharing systems (Anderson et al. 2002; Larson et al. 2009). Section 2.3 describes how result verification is done in public computing systems. Finally, Section 2.4 explains the checkpoint/restart enabling techniques(Treaster 2005; Elnozahy et al. 2002; Maloney & Goscinski 2009).

2.1 Peer-to-Peer Systems

Peer-to-peer lacks a consensual formal definition. Common users understand peer-to-peer as the type 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 (Napster 1999), KaZaA (KaZaA 2000) and BitTorrent (BitTorrent 2003) have successfully enabled users to exchange files among themselves. More generally peer-to-peer can be defined as the type 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 transient populations 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.

2.1.1 Applications

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

Distributed Computing Systems

These systems take advantage of computing power located at the edges of the network to speed-up the execution of computationally expensive tasks. In order to prevent the user's frustration due to a lack of computational power, these applications usually execute as low priority processes or only when the computer is idle, acting like screen-savers (Anderson 2004). 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 long running task. This parallelization of the computation leads to a better performance. Nevertheless, only some computational tasks are appropriate to this model of execution and even if they are, they may not have a visible speed-up if they have a low computation-transmission ratio (this ratio is explained in more detail in Section 2.2).

Examples of these are Seti@Home (Anderson et al. 2002), GIMPS (GIMPS 2010), distributed.net (distributed.net 1997), Folding@home (Larson et al. 2009). Nonetheless, some authors disagree that these are Peer-to-peer systems, for the participants are cycle farms that an institution explores. There is no actual exchange of resources, participants donate their spare cycles to a cause they regard as legitimate. These projects usually attract their participants using other incentives, Section 2.2 explains these incentives.

GINGER (Veiga et al. 2007) is a Peer-to-peer Distributed Computing System that enables the participants to exchange cycles among themselves, being every participant able to execute tasks for other participants and submit his own tasks to be executed in participants as well.

Content Sharing Systems

These 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 different parts of the same file from several users simultaneously, creating a distribution system with no bandwidth bottlenecks. Content Sharing systems enable their participants to exchange their files within a community.

The contents shared in these systems are usually digital media (e.g., music, films, books). Napster (Napster 1999) was one of the first systems that distributed music in MP3 digital audio format. KaZaA (KaZaA 2000) and BitTorrent (BitTorrent 2003) are examples of other peer-to-peer systems that enable the distribution of content through the internet.

Collaboration Systems

These 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.

2.1.2 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 from 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.

2.1.3 Network Overlay Centralization

Considering the peer, super-peer, and central server 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

These architectures are composed exclusively 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, the location of participants or resources is a challenging issue in these architectures.

Freenet (Clarke et al. 2001) uses a purely decentralized data storage system. Participants contribute with storage space to provide a non-censurable network of contents. This is possible given that these architectures have no single points of failure.

Partially Centralized Architectures

These architectures are composed by peers and super-peers. Super-peers are peers which 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 are dynamically assigned, they do not constitute single points of failure or scalability issues (if a super-peer fails, another peer is chosen to become a super-peer).

In KaZaA peers with high bandwidth become super-peers that maintain an index of the files located in a set of peers with lower bandwidth.

Hybrid Decentralized Architectures

These architectures are composed by peers and a central server. The exchange of resources is done directly between two peers. The central server performs complementary, but essential, tasks (e.g., bootstrapping, indexing of resources or participants). This central server enables an efficient location of participants and resources. Nonetheless, the central server constitutes a single point of failure and limits the scalability of the system.

In Napster 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.

Beside these three degrees of centralization, there are other systems which are considered by some authors as peer-to-peer systems. Seti@Home (Anderson et al. 2002) is on the frontier 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 advantage from resources at the edges of the Internet and therefore must 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 (peers provide a resource to a central server).

Table 2.1: Peer-to-peer degree of centralization.

	Client-server		**Peer-to-peer**		
		Inverted Client-Server			
Centralization	Centralized	Centralized	Hybrid Decentralized	Partially Centralized	Purely Decentralized
Entities	Clients and Server	Peers and Master Server	Peers and Central Server	Peers / Super-peers	Peers
Exchange of Resources	From Server to Clients	From Clients to Server	Between Peers	Between Peers (Super-peers are also Peers)	Between Peers
Examples	Web Browsers and Web Servers	Seti@home, Folding@home, GIMPS, distributed.net	Napster	KaZaA	Freenet, GINGER

Table 2.1.3 resumes the peer-to-peer systems according to their overlay centralization categories.

2.1.4 Network Overlay Structure

Peer-to-peer systems can also be classified having into consideration the way their network overlay[1] is structured. It basically defines the connections that exist among the peers in a peer-to-peer system. Systems have been built with structured, unstructured, and hybrid network overlay topologies.

Unstructured Systems

These systems create their overlay in an ad-hoc manner, not following any specific rules. A peer is connected to a random set of other peers. 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 (Napster 1999), KaZaA (KaZaA 2000) are examples of unstructured systems.

Structured Systems

These systems create an overlay that obeys 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. However, 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. Chord (Stoica et al. 2001) and CAN (Ratnasamy et al. 2001) are examples of structured systems.

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

Chord 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) 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.

Hybrid Systems

These 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. However, this naive approach would generate high overheads. Pastry (Rowstron & Druschel 2001) and Kademlia (Maymounkov & Mazières 2002) have developed more elegant techniques that conciliate structured and unstructured models.

Pastry 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 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.

2.2 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 mostly composed by non-dedicated machines. These systems emerged in institutional environments (Litzkow et al. 1988) and were later transposed to public environments (Anderson et al. 2002). 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.

2.2.1 Applications

Most of the Cycle-sharing systems developed so far focus on the execution of long running tasks related to Human causes: Folding@home simulates the folding of proteins which may lead to a better understanding and possible cure of certain diseases; Seti@home analyses radio signals in order to find evidence of intelligent extraterrestrial life. Such projects have attracted more attention than initially predicted, and show that the general public is willing to donate their spare CPU cycles. They created communities around these projects/causes through forums, ranking tables, teams, statistics, that they believe resulted as a crucial incentive to the participants.

Nonetheless, other applications are fit to the Cycle-sharing execution model like: ray-tracing, fractal generation, video transcoding. To be amenable to distributed computation, must be possible for the application to have its work partitioned in multiple tasks that execute separately. Plus, the applications must perform a considerable amount of execution over a relatively small portion of data, otherwise the transmission cost would make the computing non-profitable. This relation between the portion of data and the amount of computation is called computation-transmission ratio and impacts the size of the tasks. For example, the POV-Ray data is always the whole POV file, no matter if we want to calculate a pixel, a line or the whole image. The amount of data to be transferred for calculating a pixel would produce a low, non-profitable, computation-transmission ratio.

2.2.2 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 2.2.2 shows an inverted client-server architecture where a server communicates with a group of heterogeneous machines.

Cluster Computing On the Fly (Lo et al. 2004) 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

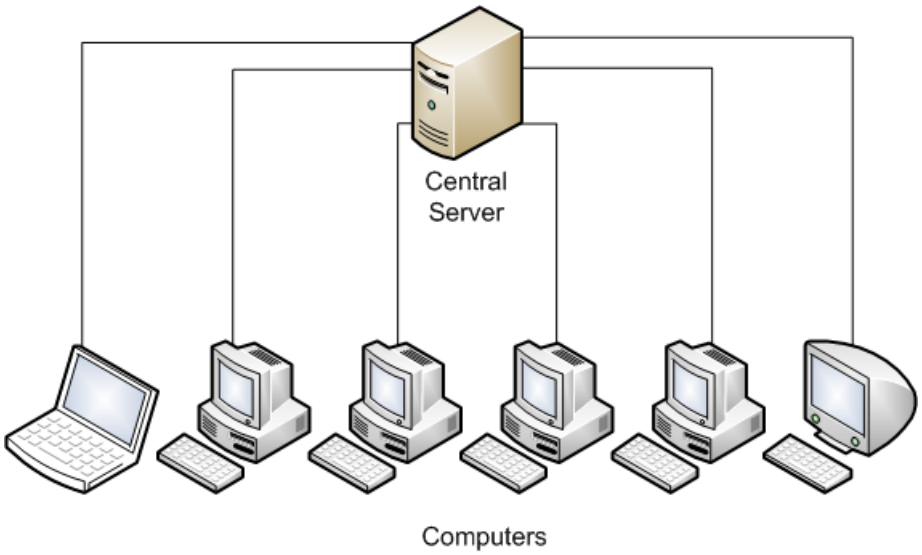

Figure 2.1: Inverted client-server architecture in cycle-sharing systems.

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

2.3 Result Verification

To speed up the execution of long running algorithms, high performance public computing systems schedule jobs to be executed in other participants. These participants may not be trusted and return wrong results. Verifying the correctness of the results is a very expensive task. An exhaustive verification of these results would cause a major slowing down of the system, defeating its purpose. Therefore, the results must be verified using an affordable reliability level.

2.3.1 Incorrect Results

Wrong results can have different motivations. Taking into account their motivations they can be distinguished into faulty (non-intentional wrong results) and malicious results (results that are intentionally forged).

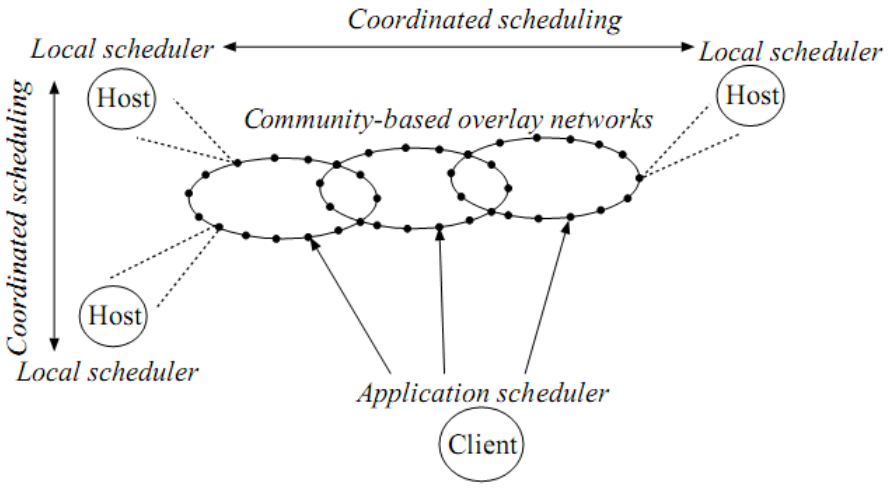

Figure 2.2: Architecture of Cluster Computing On the Fly.

Faulty Results

These results have no motivation: they are originated by faults or byzantine behaviour. Although they can generate unpredictable wrong results, both types usually produce incoherent results that are relatively easy to identify, when compared with results that are intentionally forged with malicious motivations.

Malicious Results

These results are intentionally created to harm the system. They are attacks that explore the vulnerabilities of the system, causing it to work inappropriately. These results usually fall into one of the following subcategories: cheating, isolated, collective.

Cheating malicious results are returned by cheating participants to receive credit for the work they have not performed. These are only profitable to the malicious participant if the cost of forging them is inferior to the cost of the task. The majority of the cycle sharing systems motivates participants through communities, teams, awards, and ranking tables. Some participants are willing to corrupt their results to receive credit for work they did not perform (Molnar 2000).

Isolated malicious results are forged by a malicious participant to discredit the system. This forging usually takes into account possible vulnerabilities of the result verification mechanisms

and produces a forged result at any cost.

Collective malicious results intend to harm systems that use replication as means to verify their results. To perform these, attacks several participants return the same bad result, maximizing the chances of this bad result being the one chosen in the voting quorums (Douceur 2002). This is know as collusion and requires communication between the malicious participants to identify if they are performing the same task.

2.3.2 Techniques

Several techniques have been proposed to identify bad results. Techniques vary in their complexity, overhead, and effectiveness. However, no single technique can identify all the types of bad results we have defined. Nevertheless, it has been demonstrated that these techniques, combined with a reputation system, can improve the reliability of the results produced by the system (Zhao et al. 2005).

Replication

One of the most effective methods to identify bad results is through 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, three 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 (Anderson et al. 2002) and Folding@Home (Larson et al. 2009) use redundant execution and voting quorums to verify their results.

Replication consumes, at the very least, three times 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

This technique is able to defeat cheating participants by forcing them to calculate a binary hash-tree from their results, and return it with them (Du et al. 2004). 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. Figure 2.3.2 shows a hash-tree where the leafs are partitioned sequential results or the data to be checked, 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.

This dissuades cheating participants because finding the correct hash-tree requires more computation than actually performing the required computation and producing the correct results.

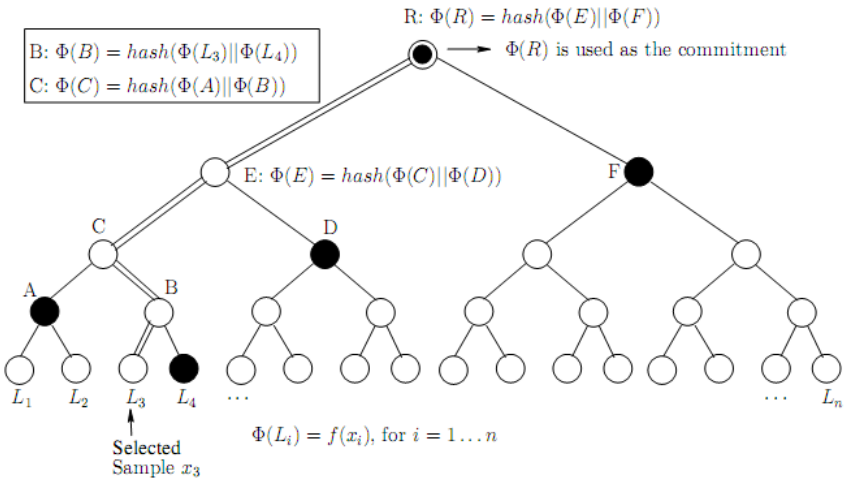

Figure 2.3: Example of an hash tree.

Hash-trees make cheating not worthwhile. They have a relative low overhead: a small portion of the work has to be executed locally and the hash tree must be checked. However, they do not dissuade malicious participants that are willing to forge their results at any cost.

Quizzes

This technique consists in assigning jobs whose result is known by the submitter a priori. Therefore, these jobs can test the honesty of a participant. Cluster Computing On the Fly (Lo et al. 2004) 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 (see 2.3.3) that manages the trust levels of the executing peers. However, 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 in 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. However, their 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.

2.3.3 Reputation Mechanisms

Reputation mechanisms (Kamvar et al. 2003; Zhao et al. 2005) are inherent to security and are beyond the scope of this work. Nonetheless, they can improve the reliability of the results by influencing the scheduler, preventing the assignment of work to participants that are considered malicious or least trusted (e.g., blacklisting, ranking). In order to do so, the reputation mechanism relies on the information provided by the result verification mechanisms. Although the result verification mechanisms accept results that are believed as correct within a predetermined probability, they can sometimes identify wrong results with certainty. Therefore, the better the information provided by the result verification mechanisms, the better the reputation mechanisms effectiveness.

2.4 Checkpoint/Restart

Checkpointing is a primordial fault-tolerance technique. Long running applications usually implement checkpointing mechanisms to minimize the loss of work already performed 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 (Cezário & Sztajnberg 2008) is the resuming of an application that was checkpointed elsewhere (on another machine) and can improve the performance of a high performance distributed computing system that is composed by dynamic heterogeneous participants.

2.4.1 Implementation Approach

To provide an application with checkpoint/restart capabilities, three approaches must be taken into account: Application-level, Library-level, and System-level.

Application-level Checkpoint/Restart Systems

These systems are built within the application code requiring a big programming effort. If the applications are not developed from scratch to support checkpointing mechanisms, it may be impossible to provide them with checkpoint/restart capabilities later. Since the programmer knows exactly what needs be safely stored to enable the application to restart in case of failure, application-level checkpoint/restart are usually more efficient. They achieve a better performance, and lower checkpoint data size. Applications may either checkpoint at time intervals, or constantly persist the important data. Since these systems do not use any operating system support, they are portable[2].

However, the previous 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

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

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, and exhausting, 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).

Seti@home (Anderson et al. 2002) and folding@home (Larson et al. 2009) use this implementation approach, the checkpointing mechanisms are built within their algorithms.

Library-level Checkpoint/Restart Systems

These consist in linking a library with the application, creating a layer between the application and the operating system that provides checkpoint/restart capabilities. The major advantage is that it is possible to create generic checkpointing mechanism that is able to checkpoint a vast range of applications without having to modify them, while maintaining portability.

However, the existing implementations are not able to checkpoint a vast range of applications, the major challenge is that these systems cannot access kernel's data structures (e.g., file descriptors), so this layer has to emulate operating system calls. This layer has no semantic knowledge of the application and checkpoints may be taken in the least appropriate moments generating considerable sized checkpoints. This layer may also be responsible for a slowdown in the performance of the application.

Libckpt (Plank et al. 1995) implemented 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 only able to provide checkpoint/restart capabilities to a limited scope of applications because it cannot access system states maintained by the kernel.

The Condor distributed processing system (Litzkow et al. 1997) studied the possibility of using these mechanisms to provide several applications with checkpoint/restart and even migration capabilities.

System-level Checkpoint/Restart Systems

These systems are built as an extension of the operating system's kernel. They are more powerful, since they can access kernel's data structures (e.g., file descriptors). Checkpointing can consist in flushing all the process's data and control structures to stable storage (i.e., to a file on the local disk). 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, so they are transparent to the application.

These approaches have the disadvantage of not being portable. The non-knowledge of the application semantics leads to least efficient checkpoint data when compared with checkpoint data generated by applications that checkpoint themselves (i.e., application-level). Plus, developing a kernel module that enables the checkpointing of any application is complex and the implementations so far are only able to checkpoint some applications.

CRAK (Zhong & Nieh 2002), Checkpoint Restart As a Kernel module, is a Linux kernel module that implements mechanisms that enable the checkpoint/restart of any application. 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 being able to provide any application with checkpoint/restart capabilities.

We have described three approaches to provide an application with checkpoint/restart capabilities. Library-level and System-level are valid approaches, the current deployments are only able to checkpoint/restart a limited number of applications though. Application-level is used in various genres of applications, not long running algorithms only, in which is desirable to save state in case a fault occurs.

2.4.2 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 Checkpoint/Restart Systems

These systems try to find a match between the checkpoints taken by each of the processes to create a global checkpoint (Elnozahy et al. 2002). 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 domino effect[3] (Baldoni et al. 1995); 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.

Coordinated Checkpoint/Restart Systems

In these systems, processes cooperate to create a global consistent checkpoint (Elnozahy et al. 2002; Chandy & Lamport 1985). 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 their 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 Checkpoint/Restart Systems

These systems combine coordinated and uncoordinated checkpointing methods in order to avoid the domino effect and allow processes to checkpoint more autonomously (Elnozahy et al. 2002). The method 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] 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.

2.4.3 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. Three logging mechanisms have been proposed: Pessimistic, Optimistic, and Causal.

Pessimistic Message Logging

These logging mechanisms log 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 (Elnozahy et al. 2002). 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

These systems log the event to volatile storage instead of stable storage (Elnozahy et al. 2002). 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

These systems try to take advantage of the previous methods (Elnozahy et al. 2002): 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 method and avoids the loss of events of the optimistic method. However, events may still be lost due to the failure of several processes or applications.

2.4.4 Enhancements

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

Concurrent Checkpointing

Concurrent Checkpointing aims at reducing the time a process is blocked due to a checkpoint operation (Elnozahy et al. 2002). While the process is being checkpointed, it remains blocked so it cannot modify its memory. Concurrent checkpointing reduces the time a process remains blocked by marking its memory copy-on-write. This allows the process to be unblocked during the checkpointing.

Incremental Checkpointing

This technique avoids rewriting portions of the process state that did not change between checkpoints (Feng & Lee 2006; Lawall & Muller 2000). 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 extreme cases 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

This technique uses volatile memory to store checkpoints which provides decreased storage times (Plank et al. 1998). 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).

In this Chapter we have described Peer-to-peer systems: their applications; their architectures; and how they can be categorized taking into account their network overlay centralization and structure. We have introduces Cycle-sharing systems: their applications and architectures. We have described the state of the art result verification and checkpoint/restart mechanisms. In the next Chapter we describe our architecture.

3 Architecture

In this Chapter we describe our architecture. Section 3.1 presents the overall architecture of GINGER. Section 3.2 describes the model of faults and non-regular behaviour that we must tolerate and handle. Section 3.4 describes two complementary techniques that provide applications with checkpoint/restart capabilities. Section 3.3 details a number of result verification strategies.

3.1 Architecture of GINGER

GINGER combines Peer-to-peer and Cycle-sharing technologies to enable every user to donate their spare cycles, and use the spare cycles of other users as well.

GINGER's middle-ware is able to run a vast range of unmodified desktop applications that are fit to this model of computing, such as digital media compression/transcoding/enhance and content generation (e.g., ray-tracing, fractal generation). In order to be able to run these unmodified applications, GINGER proposes the concept of a Gridlet, a semantics-aware unit of workload division and computation off-load (basically, a chunk of data and the operations to be performed over it).

Figure 3.1 depicts a global view of the GINGER architecture. The GINGER layered middle-ware runs in each of the participants, this layered architecture enables portability and extensibility. Next, each of the layers is described:

- Application Adaptation Layer: is responsible for interacting with the actual unmodified desktop applications, e.g., launch them, feeding the data inside gridlets, and collecting results. This is the specific application adaptation layer, extending GINGER to support new applications is done within this layer;

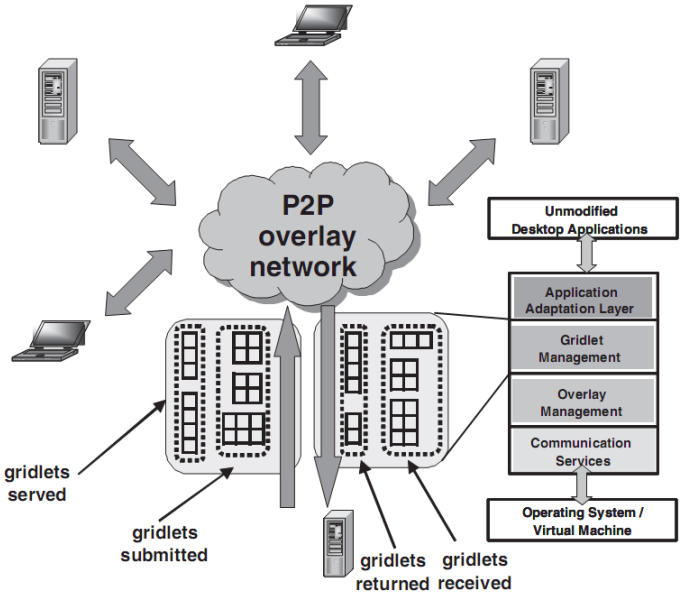

Figure 3.1: Architecture of GINGER.

- Gridlet Management Layer: performs the tasks necessary to partition the long running execution into several gridlets and reassemble the results later (using the above layer for application specific details), this layer is also responsible for scheduling the task to available participants;
- Overlay Management Layer: is responsible for maintaining the overlay network to exchange gridlets with other nodes. Maintaining the overlay consists in the constant discovery and analysis of the presently available resources;
- Communication Services Layer: is responsible for carrying out the actual network transfer.

Result verification and checkpoint/restart enabling mechanisms are part of the Gridlet Management layer. The result verification strategies and checkpoint/restart policies are completely independent from the applications. They interact with the Application Adaptation Layer for gridlet creation and retrieval of the results once they correctness has been checked.

3.2 Fault Model

Public environments are composed by users without low IT know-how, heterogeneous hardware and network conditions (possibly more prone to failures) and untrusted participants (possibly malicious). In order to improve the robustness of GINGER, we want to support various faults and non-regular behaviour.

Checkpoint/restart mechanisms trigger the rescheduling of a new task that either resumes from the last checkpoint or starts from the beginning of the task, if no checkpoint was taken during fault-free execution, in any of the following scenarios:

- the participant makes an announced departure during the execution of a task;
- the participant fails the reply to a ping call (silent participants are considered failed);
- the participant takes too long to retrieve a checkpoint or a result (even if it replies to the ping call, if it does not retrieve a checkpoint or result within a predetermined time interval, it is considered failed);

Result verification mechanisms must be able to attenuate the impact of wrong results, providing additional reliability to the results produced by the system. They must be able to detect bad results originated by faults and malicious behaviour, either isolated or collective, with a predetermined probability. If a result is considered incorrect by the system, the system must be able to reschedule the corresponding task to another participant until a result that is considered as correct is retrieved.

3.3 Result Verification

In order to accept the results returned by the participants, we propose a number of replication approaches with some extra considerations, a complementary sampling technique, and the merging of both techniques.

3.3.1 Incremental Replication

The insight of assigning the work iteratively, according to some rules, instead of putting the whole job for execution at once can provide some benefits with only minor drawbacks.

As an example of a rule, we can assign only the required redundant work for the voting quorums to be able to accept it. The major benefit stems from the fact that a considerable amount redundant execution is not even taken into consideration when the correct result is being chosen by the voting quorums. For example, for replication factor 5, if 3 out of the 5 results are equal, the system will not even mind looking at the other 2 results. Then, those could and should never have been executed. And if so, the overall execution power of the system would have been optimized by avoiding useless repeated work.

Another example of a rule: no redundant work is ever in execution at the same time. This has benefits in colluding scenarios. In these, the same bad result is only returned once the colluders have been able to identify that they have the same job to execute. If a task is never being redundantly executed at the same time, colluders can only be successful if they submit a bad result and wait for the replica of that task to be assigned to one of them, enabling them to return the same bad result. If that does not happen, the bad result submitted by them will be detected and they may be punished by an associated reputation mechanism (e.g., blacklisted). This would force the colluders to maintain records of the gridlets assigned to them during a period of time and periodically exchange that information for collusion to be successful.

This technique can have a negative impact in terms of time to complete the whole work: on the one hand, the incremental assignment and wait for the retrieval of results will lower the performance when the system is not overloaded; on the other hand, if the number of available participants is low, it can actually perform faster than putting the whole work for execution at once. Therefore, the correct definition of an overloaded environment, taking into consideration various factors (e.g., the number of available participants, the maximum number of gridlets, etc.) makes possible for the system to decide whether to use this technique or not, enabling it to take the best advantage of the current resources.

3.3.2 Replication using Overlapped Partitionings

Using overlapped partitioning, the tasks are never exactly equal, even though each individual piece of data is still replicated with the predetermined factor. Therefore, it becomes more complex for the colluders to identify the common part of the task, plus they must always execute part of the task, even when they are trying to return forged results. Figure 3.2 depicts the same work divided in in two different overlapped partitionings.

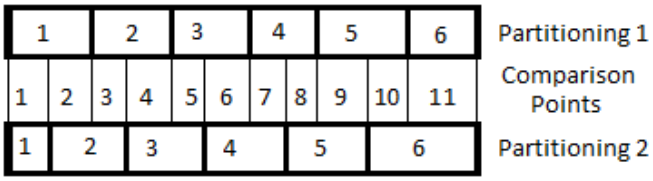

Figure 3.2: The same work divided differently, creating an overlapped partitioning.

These overlapped partitionings can use a random offset and require strong communication among the colluders to identify the common part of the job. Although it is more probable for colluders to have common parts of the tasks (since, for replication factor two, every task has redundant parts in two other tasks, instead of one as in standard partitioning), these common parts are smaller (part of a task, instead of the whole task).

3.3.3 Replication using Relaxed Partitionings

Overlapped partitioning can be implemented in a relaxed flavour, where only some parts of the job are executed redundantly. This lowers the overhead, but also lowers the reliability of the results. However, it can be useful if the system has low computational power available. The behaviour of the malicious participants would be tricky, they are able to detect the common part of the job, however they can never be sure that the non-common part is not being executed redundantly. Figure 3.3 depicts a relaxed overlapped partitioning.

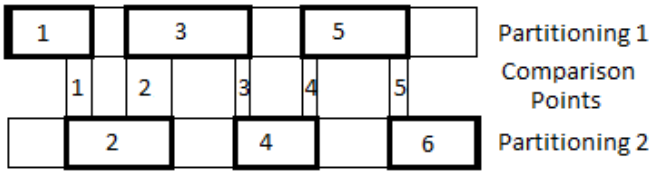

Figure 3.3: Overlapped tasks for relaxed replication.

3.3.4 Replication using Meshed Partitionings

Some applications can have their work divided in more than one dimension. Figure 3.4 depicts the partitioning of the work for a ray-tracer. Like the overlapped partitioning, it influences the way colluders are able to introduce bad results: more points where they can collude, with a smaller size too. This partitioning provides a number of points of comparison. This information feeds an algorithm that is able to choose correct results according to the reputation of a result, instead of using voting quorums.

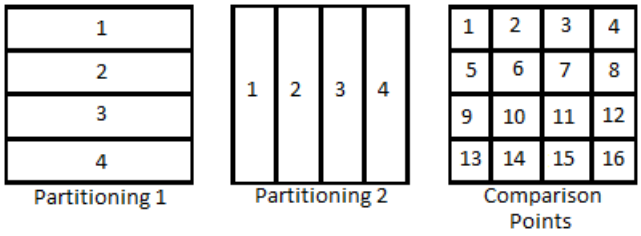

Figure 3.4: Meshed partitioning using replication factor 2.

The algorithm for calculating the reputation of a result must take into account the comparison points result (i.e., equal or not equal). Since the majority of the participants is expected to be honest equal adds positive reputation and not equal adds negative reputation.

For the acceptance of the results, equal results are accepted on the fly (if at least one of the tasks has a positive reputation), different results are disambiguated according to the reputation of the two results. If the reputation is drawn, the portion of execution of the corresponding result must be re-executed for voting quorum like disambiguation.

Figure 3.5 depicts the comparison point results of a two-dimensional work partitioned in four independent tasks twice (replication factor 2) creating eight tasks (H1, H2, H3, H4, V1, V2, V3 and V4). Using the comparison point indexes of Figure 3.4: results 1, 2, 3, 5, 6, 7, 13,14 and 15 would be accepted on the fly for being equal; in position 9 the chosen result is the one returned in task V1 since it has reputation 2 against reputation -4 of task H3 (positions 10, 11, 4, 8 and 16 would be disambiguated in the same way); position 12 would require re-execution, since both tasks H3 and V4 have equal negative reputation -4.

This algorithm can enable the use of low, possibly even, replication factors. Requiring minimal portions of extra execution for disambiguation. Furthermore, in a system where the majority of the participants is honest, the extra work is minimal and rare.

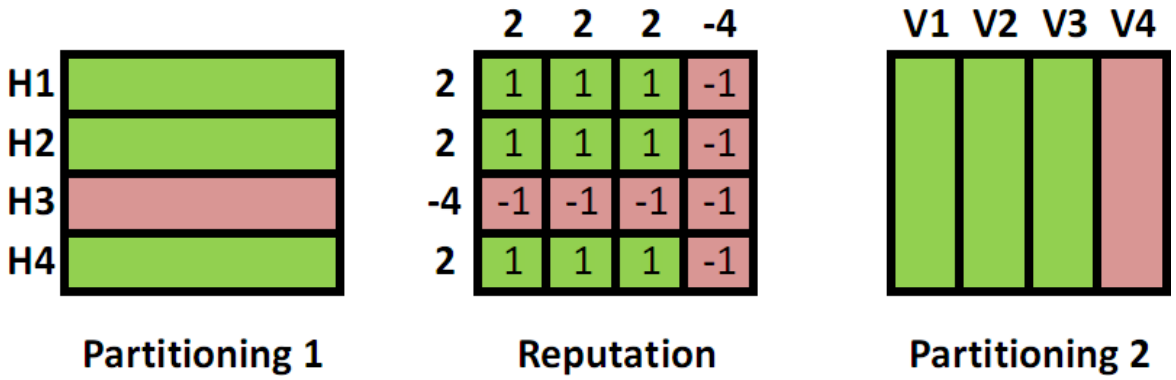

Figure 3.5: Meshed partitioning: results of the comparison points (1 means equal, 0 means not equal).

Bi-dimensional tasks consume twice the resources (as replication factor 2), plus the re-execution of small portions of work that were received but are either impossible to disambiguate or considered incorrect. In essence, it consumes the same base amount of work as incremental replication with replication factor 3 (assign only enough work to win the quorums), but the amount of results that require rescheduling is lower.

3.3.5 Random Sampling

Replication bases all its result verification decisions in results/info provided by third parties, i.e., the participant workers. In an unreliable environment this may not be enough. Therefore, local sampling can have an important place in the verification of results. Figure 3.6 depicts the sampling of an image where a sample is a pixel.

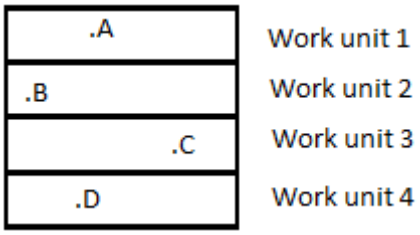

Figure 3.6: Sampling for an image.

Sampling considers the local execution of a fragment, as small as possible, of each task to be compared with the returned result. In essence, sampling points act as hidden embedded quizzes without being hindered by the generation nor the identification issues described in the related work. Sampling also enforces that the malicious participants execute part of the task for

this to have any chance of being accepted. Although random sampling can only ensure that a result is correct with a given probability (based on the size of the work, the number of samples and the percentage of the work that is corrupted), it can identify wrong results with certainty and deliver very useful information to a reputation mechanism.

3.3.6 Samplication

Replication and random sampling can be used sequentially to achieve higher reliability of the results: the winning result of the voting quorums is considered correct if it matches a random sample that was executed by the submitter. Furthermore, these techniques can be combined in a more elegant manner that provides additional benefits without adding extra overhead.

The algorithm we propose is very simple: the next piece of pseudo-code describes how it works:

```
 1.  Schedule redundant work, put the results in a bag;
 2.  IF(the bag is empty)
 3.    GOTO 1;
 4.  IF(all results in the bag are equal)
 5.    IF(random sample matches)
 6.      ACCEPT RESULT;
 7.    ELSE
 8.      remove all results from the bag;
 9.      GOTO 1;
10.  ELSE
11.    choose a sample within the mismatch area;
12.    compare with all results;
13.    remove results that mismatch the sample from the bag;
14.    GOTO 2;
```

This algorithm provides a number of desirable properties:

- It only discards wrong results: if at least one of the results of the redundant work is correct, this approach ensures that the mentioned result is the chosen one, and that the

honest participant will always receive credit for it, whereas using voting quorums, if the correct result is not within a majority it is discarded, the honest participant does not receive credit and might even be punished by the reputation mechanisms.

- It enables the identification of fault-prone participants and colluders: results that mismatch samples are wrong, if more than one result are equal and wrong it is very likely that they were returned by colluders;
- It enables the use of even replication factors (since it does not use voting quorums);
- The number of samples per task is low: if all results are correct, it only requires one sample per task; the maximum number of samples used when there is no rescheduling is R (R being the replication factor);
- Rescheduling only occurs if all the received results were wrong, which makes rescheduling the desirable option.

To improve the reliability of the results, we have proposed various result verification schemes. Incremental replication can reduce the overall amount of work required in the voting quorums. Overlapped partitionings difficult the colluders identification of the redundant work. Relaxed partitionings are a viable option for scenarios with low computational power available. Meshed partitionings employ stateless reputation mechanisms to decide the result that is accepted and enable even replication factors. Samplication is an algorithm that makes elegant use of replication and sampling, enables even replication factors, and detects faulty/malicious participants and colluders.

3.4 Checkpoint/Restart

In GINGER we want to provide a wide range of applications with checkpoint/restart capabilities, while keeping them portable to be executed on cycle-sharing participant nodes that use different platforms, and without having to modify them. Library-level is the only approach in the related work that would fit. However, an approach simply stating these goals is still far from being able to checkpoint a wide range of applications, and would require the recompilation or replacement of libraries in the executing peers. Therefore, we propose two mechanisms that will enable us to checkpoint/restart any application. Our first approach is through

a virtual machine, which has some efficiency drawbacks but is able to checkpoint/restart any application. Furthermore, we propose checkpoint/restart through the results, that will enable some applications to checkpoint very efficiently.

3.4.1 Through a Virtual Machine's Running Image

An application can be checkpointed if we run it on top of virtual machine with checkpoint/restart capabilities (e.g., qemu (QEMU 2010), VirtualBox (VirtualBox 2010)), the application's state being saved within the virtual machine's state. This also provides some extra security to the clients, since they will be executing untrusted code with a high level of confinement.

The major drawback of this approach is the size of the checkpoint data, incurring considerable transmission overhead. To attenuate this: 1) we assume that one base-generic running checkpoint image is accessible to all the peers; 2) the applications start their execution on top of this image once it is locally resumed; and 3) at checkpoint time we only transmit the differences between the current image and the base-image. Since the base image has to exist for every participant, but it is never transmitted between participants it can be considered part of the GINGER application.

Virtual disk formats fully support this differencing disk functionality along with compression (McLoughlin 2008), producing optimized differencing disk files that can be quickly transmitted whereas the overhead of the transmission of the complete image for each checkpoint would be unbearable.

Since we are transmitting a running image, we have to transmit not only the virtual disk but also the volatile state. Differencing the volatile state is also an option, although there is no actual deployment of differencing volatile states, several efficient differencing algorithms have been proposed and can be used (Burns et al. 1997).

The checkpoint data size can be further reduced using optimized operating systems (just enough operating system or JeOS), that boot only a small subset of services required to run the applications, leading to smaller disk images and volatile states. And, finally, over all these techniques compression is still an option to further reduce the overhead.

This approach does not have semantic knowledge of the applications and it cannot preview

results. However, we may be able to show some statistical data related to the execution and highlight where changes have occurred. Checkpoints would be taken within predetermined time intervals.

3.4.2 Through the Result Files

This technique will only be fit for some applications and demands the implementation of specific enabling mechanisms for each application, although without requiring modifications to its code. The idea behind this technique is that the applications produce final results incrementally during their execution. Therefore, if we are able to capture the partial results during execution and resume execution from them later, such result files can actually serve as checkpoint data. This creates a very efficient checkpointing mechanism for the results have to be transmitted later anyway.

This technique can be implemented using two different approaches: by monitoring the result file that is being produced by the application; or by dividing the gridlet work into subtasks in the executing peer. Monitoring the file while it is being written requires a daemon process and requires the application to write the result to a file during, rather than, at the end of the execution. The division in subtasks requires the invocation of the application several times, which may lead to a small delay in the overall execution. This division has to be done in the executing peer, breaking it in the submitter (i.e., create smaller gridlets) can lead to a non-profitable computation-transmission ratio.

Since this approach has semantic knowledge of the application's result it can checkpoint whenever it is more convenient for the application (e.g., every 10 lines in an image written by a ray-tracer); rather than at a predefined time interval. This awareness of the semantics of the application also enables the monitoring of the execution's progress and the previewing of the results in the submitter.

In this Chapter we have described GINGER's architecture, and we have defined our fault model. For a more reliable result verification we have proposed incremental replication, replication with overlapped, relaxed and meshed partitionings, random sampling and a technique that combines replication and sampling in an elegant manner. For mitigation of the volatile nature of the participants we have proposed checkpoint/restart through a virtual machine's

running image and through the result files. In the next Chapter we describe our implementation.

4 Implementation

In this Chapter we describe our implementation. Section 4.1 describes our simulator. The simulator enables us to perform statistical analysis of our result verification approaches behaviour with large populations. Section 4.2 describes our real deployment, that proves that our result verification and checkpoint/restart approaches are feasible.

4.1 Simulator

The simulator is a simple, non-communicant Java application that simulates a scenario where an n-dimensional long running task is broken into work-units that are randomly assigned. Among the participants, there is a number of malfunctioning participants that retrieve wrong results and also a group of colluders that attempt to return the same bad result (based on complete or imperfect knowledge, depending on the partition overlapping), in order to fool the replication-based verification mechanisms.

The simulator returns several metrics that allow us to analyse the behaviour of our result verification strategies in different environments. The most important among the results returned by the simulator is the percentage of bad results that were accepted by the system. Since the simulator uses a random scheduler, the results returned by the simulator may vary. Therefore, to accurately determine a result, it must be the mean of several simulations (e.g., 2000 simulations). Figure 4.1 depicts the evolution of the mean of the percentage of bad results accepted through to 10000 simulations. This confirms the significance of the obtained results, since in a real deployment, the scheduling of the gridlets would also produces unpredictable variations.

In order to run a simulation, we must set a number of options and parameters that define:

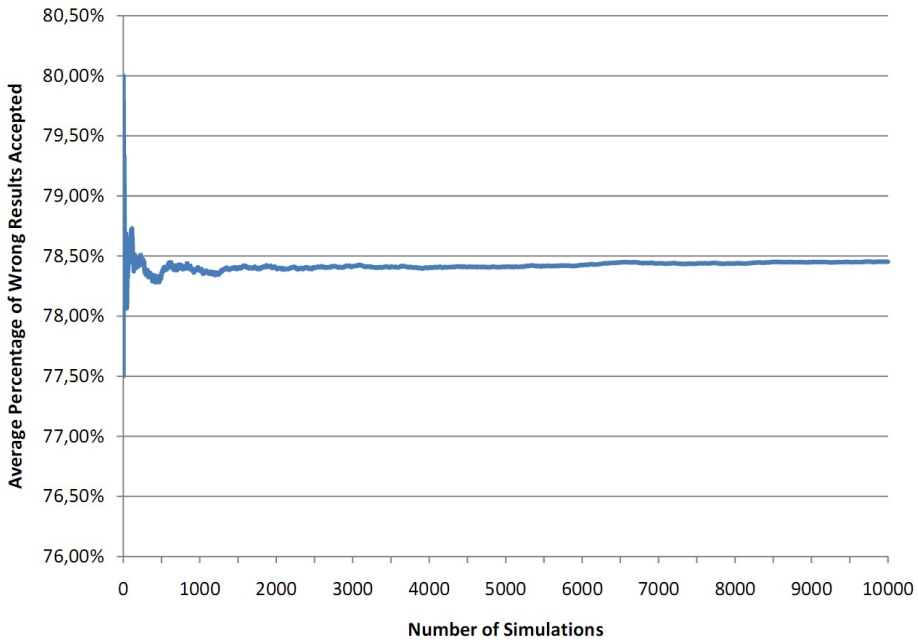

Figure 4.1: Evolution of the mean of the percentage of bad results accepted through 10000 simulations.

the participants, the task, and the result verification strategy. These options and parameters are described next.

4.1.1 Participants

For each simulation we must set the environment (i.e., the community of participants that will execute the tasks). For that, we define the following parameters:

- Number of Participants: the total number of available participants of the system (e.g., 1000);
- Number of Colluders: how many of the participants are part of a group of malicious communicant participants (e.g., 50). These participants return the same bad result if they are executing redundant work. We also have to define the percentage of work that is colluded (e.g., 50%), enabling the simulation of imperfect identification of the common work (and the maximizing of the chances of acceptance of a wrong result if the result verification technique is sampling, for example);
- Number of Bad Workers: how many of the participants will retrieve isolated bad results,

simulating either isolated malicious or faulty behaviour (e.g., 100). The percentage of work that is incorrect must also be defined (e.g., 75%).

4.1.2 Task

We must set the work size of the task. The size is defined in terms of atoms of execution (i.e., an indivisible portion of execution), an atom of execution of an image generation application is the amount of execution that produces a pixel, for example. Tasks can be n-dimensional, since they can be represented as an array of integers. If we are computing an image with resolution 800x600, our linear (one-dimensional size) would be 480000 atoms of execution (pixels), as to our 2-dimensional approach it would be (800, 600).

The number of work units in which the long running task will be divided shall also be provided, for the simulator to be able to partition the work.

4.1.3 Result Verification Strategy

The simulator allows the choice of several result verification strategies and corresponding parameterization. They are:

- Quorum based Replication: allows the simulation of replication with standard or overlapped partitionings (we must also provide the replication factor (e.g., 3));
- Meshed Replication: allows the simulation of our algorithm for meshed partitionings (we have to provide an n-dimensional task);
- Random Sampling: allows us to simulate a stand-alone sampling strategy (we must provide the number of samples per work unit);
- Replication and Sampling: allows us to simulate a standard quorum based replication followed by a sampling technique (i.e., the result chosen in the voting quorums is further compared with a random sample, and is accepted in case it matches);
- Samplication: allows us to test our samplication technique (we must provide the replication factor).

4.1.4 Simulation Results

The simulator returns several results. Some results only make sense for some result verification strategies. It returns:

- The amount of work actually performed: it considers the amount of work that is rescheduled in some result verification schemes;
- The number of samples performed: the number of samples is variable in some techniques;
- The amount of wrong and right results accepted;
- The number of gridlets assigned to colluders, faulty/malicious and normal participants;
- The number of faulty/malicious and colluders that were detected.

4.1.5 Example of a Simulation

To run a simulation we define the task, the participants and the result verification strategy, and then we can launch the simulation by invoking the method *simulate*. The following is an example of the code to run a simulation.

```
Simulator sim = new Simulator();

// Participants
sim.setnParticipants(1000);
sim.setnColluders(700);
sim.setnFaulty(0);
sim.setPercentageWorkForgedByColluded(50);
sim.setPercentageWorkForgedByFaulty(0);

// Result verification strategy
sim.setReplicationFactor(3);
sim.setnSamples(0);
sim.setVerificationMode(Simulator.VerificationStrategy.REPLICATION_QUORUM);
```

```
// Task
sim.setnJobs(100);
int[] worksize = {10000};
sim.setWorkSize(worksize);

// Launch
SimulationResult simRes = sim.simulate();
simRes.printResults();
```

The simulator returns an object *SimulationResult*, which contains all the results. Each result can be accessed independently through *get* methods, or we can print all the results. The code above produced the following results:

```
*** SIMULATION RESULT ***
 * Size of the Task: 10000
 * Size Actually Performed: 30000 (300.0%)
 * Number of Samples Performed: 0
 * Size Bad Results Accepted: 8000 (80.0%)
 * Size Good Results Accepted: 2000 (20.0%)
 * Number of Gridlets assigned to Normal: 92 (30.666668%)
 * Number of Gridlets assigned to Faulty/Malicious: 0 (0.0%)
 * Number of Gridlets assigned to Colluders: 208 (69.333336%)
 * Number of Faulty/Malicious detected: 0
 * Number of Colluders detected: 0
```

4.2 Deployment

The deployment of this work is developed in two separate Java communicant applications, the submitter and the executer. Having symmetric participants consists in launching both the applications, or integrate them into one application. This implementation focuses on checkpoint/restart through the results and result verification, while abstracting other project related areas (e.g., resource discovery, scheduling, etc.).

To boot this system, we launch at least one submitter. The submitter boots itself, and creates an interface where executers can register their location. Next, we launch a number of executers providing them with at least one submitter's location, the executer contacts the submitter and provides its location (this abstracts resource discovery). The executers create an interface where submitters can test their availability, an executer is available if it is not executing. Submitters schedule work to the first available executer (this abstracts scheduling algorithms).

Both the executer and the submitter interfaces implement a ping method that enables failure detection. If an executer has no reachable submitters, it quits. If a submitter determines that an executer is unreachable: a) if the executer was not executing for that submitter it is just deleted from the list of executers; b) if it was executing, the submitter creates a recovery gridlet (containing the remaining work, since the last checkpoint) and reschedules it, and finally the submitter is deleted from the list of executers.

The major concern in implementing the checkpoint/restart and result verification techniques is to keep the checkpoint/restart and result verification policies separated from the application-specific adaptors. For being able to support a new application, we have to develop application specific adaptors, which consists in specializing three classes: an Application Manager, a Gridlet, and an Atomic Result.

4.2.1 Application Manager

The Application Manager is responsible for dividing a long running execution into several executable gridlets and reunite their results. For some applications, it may also enable the user to preview the results as they are received (e.g., an image being incrementally produced by a ray-tracer). Using our checkpoint/restart method through the result files approach, it is possible, not only to preview the tasks that already completed their execution, but also the ones that are currently being executed.

It is necessary to extend the abstract class *ApplicationManager* and implement a constructor and three other methods. The following code excerpt is the POV-Ray's application manager.

```
class PovRayManager
    extends ApplicationManager {
```

```
  PovRayManager(String command)
    throws ApplicationManagerException { /* */ }

  int calculateWorksize() { /* */ }

  Gridlet createGridlet(int offset, int worksize) { /* */ }

  void submitResults(int offset, AtomicResult[] res) { /* */ }

}
```

The constructor receives a string as argument. This string is the command that invokes the application (this is a simplification of the GINGER application invocation, used for this work only).

For the generic application management to be able to partition any task, it must have access to the total size of the long running task, this can only be calculated by the specific application adaptors. Therefore, it must implement the method *calculateWorksize* that retrieves the total size of the long running task, in terms of atoms of execution.

The *createGridlet* method receives the offset and the size of the task and returns a gridlet that matches the corresponding portion of execution. Gridlets are created with a size defined by generic application adaptors rather than the specific ones. This creation of gridlets on demand enables:

- the implementation of application-independent partitioning policies, manipulating the offset and worksize parameters;
- the creation of samples to be locally executed (a sample is a gridlet whose work consists in an atom of execution);
- the creation of recovery gridlets, for both checkpoint/restart and result verification purposes.

The *submitResults* method receives an offset and a variable number of ordered results in an array. This method reassembles the results, and since it receives the updated results during the execution, it can display a preview of the results already received.

4.2.2 Gridlet

The gridlets are created on demand by invoking the *createGridlet* method on a specific application manager. Gridlets are aware of their own execution, and they perform it upon the invocation of the method *execute*. The following is an excerpt of the Pov Ray's gridlet class.

```
class PovRayGridlet
    extends Gridlet
    implements Serializable, Runnable {

  AtomicResult[] execute(int offset, int worksize) { /* */ }

}
```

The specialized Gridlet class must implement the Serializable and Runnable interfaces. This enables transport by the Java RMI and allows it to perform a threaded execution at its destination. Implementing the Runnable interface requires the implementation of the *run* method (this method is implemented in the super class). The *run* method invokes the *execute* method manipulating its arguments, which enables the capture of partial results (making sequential invocations of the application). These results are used as checkpoint data in our checkpoint/restart through the result files based approach.

The *execute* method receives as argument the portion of execution to be executed (this is defined through the offset and worksize parameters), the portion of execution to be checkpointed is contained within the boundaries with which the gridlet was firstly created.

4.2.3 Atomic Result

The atomic result is just a container of result data (e.g., a pixel for image generation, a frame for video enhancing). The atomic result class must specialize a method that enables the comparison with another result. The following piece of code is an excerpt of the PovRay's Atomic Result.

```
class PovRayAtomicResult
    extends AtomicResult
```

```
    implements Serializable {

  boolean isEqual(Object obj) { /* */ }

}
```

The *isEqual* must be able to compare atomic results. For result verification purposes, the comparison of results in replication schemes using standard partitionings can be done byte-wise over raw data. However, this does not work for the other types of partitionings that we described in the architecture, nor for the comparison of samples. The implementation of this simple method enables all the result verification techniques studied in our work, while keeping their policies transparent to the application specific adaptors.

Specializations of the *Result* class must implement the Serializable interface, for the Java RMI mechanisms to be able to transmit these atomic results back to the submitter.

In this Chapter we have described our twofold implementation: the simulator and the deployment. In the next Chapter we evaluated the result verification techniques and the checkpoint/restart mechanisms proposed earlier.

5 Evaluation

This Chapter presents the evaluation of the result verification and checkpoint/restart that were proposed in Section 3. Mechanisms for fault-tolerance and reliability improve the robustness of system at a cost. The major concern in our evaluation is the overhead incurred by these mechanisms. Section 5.1 presents the evaluation of result verification mechanisms, and Section 5.2 the evaluation of checkpoint/restart mechanisms.

5.1 Result Verification Mechanisms

Every result verification technique ensures that a result is correct with a predetermined probability. This probability usually grows along with the overhead incurred by the technique. Therefore, any technique that minimizes the overhead and maximizes the probability of a result being correct is a step towards a more efficient (if it minimizes the overhead, maintaining the probability) and more effective (if it maximizes the probability, maintaining the overhead) result verification.

5.1.1 Replication

Standard replication using voting quorums can be fooled if a group of colluders determines they are executing redundant work and agree to return the same bad result, forcing the system to accept it.

The graphic in Figure 5.1 depicts that when the percentage of colluders is under 50%, the greater the replication factor the lower the percentage of bad results accepted; when the percentage of colluders is above 50%, albeit a less probable scenario, replication actually works against us.

Groups of colluders are usually expected to be minorities. However, we must take into account that if they are able to influence the scheduler by announcing themselves as attractive executers, the percentage of bad results could even be above what this graphic shows, for the scheduler it uses is random.

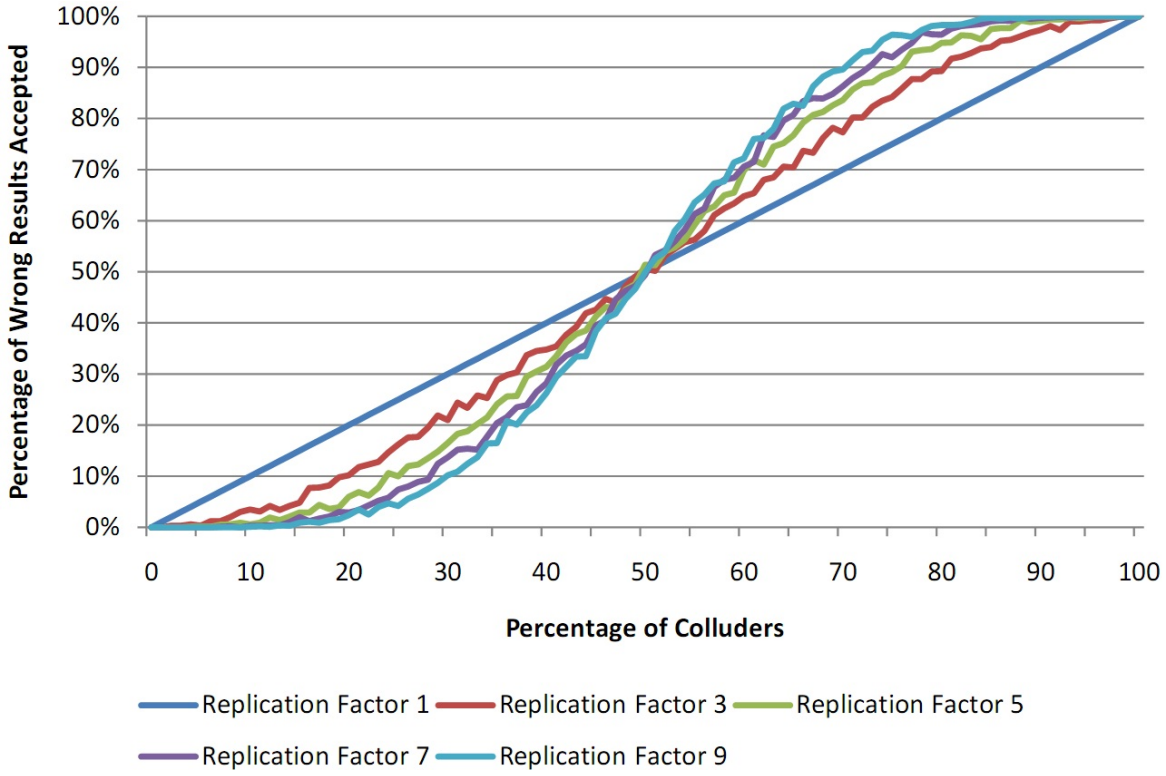

Figure 5.1: Correlation between the percentage of bad results accepted and the percentage of colluders in the system for various replication factors (colluders return results 100% forged).

5.1.2 Incremental Replication

Incremental replication assigns the work iteratively according to some rules, instead of putting the whole work to execution at once. One of the rules that can be used is based on the insight that assigning only the minimal amount of work necessary to win the voting quorums can reduce the overall amount of execution required to accept a result in the replication mechanisms, while maintaining the probability that the accepted results are correct.

The graphic in Figure 5.2 depicts a scenario where only the minimal amount of work necessary to win the voting quorums is assigned. Among the participants there is a group of colluders that always return the same wrong result. If any of the results mismatches, the sys-

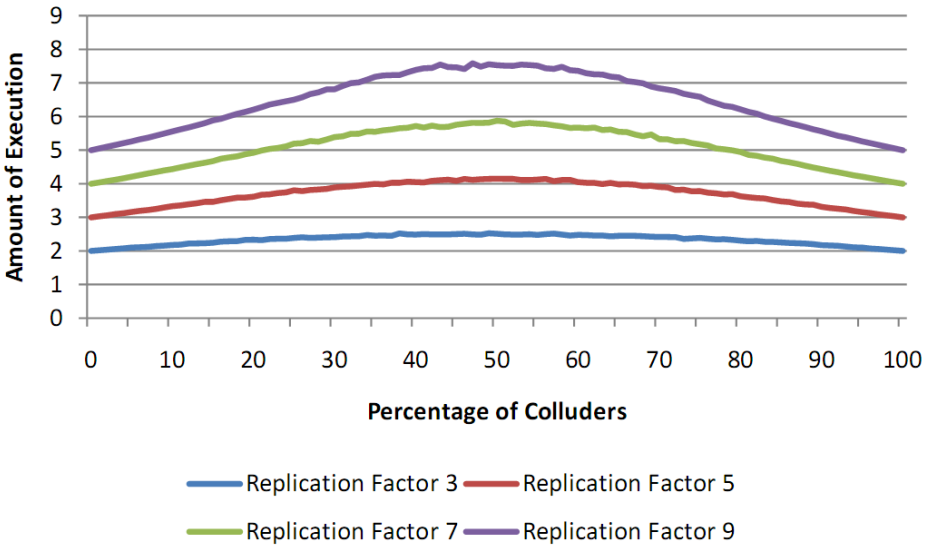

Figure 5.2: Amount of work performed using incremental replication with replication factors 3 to 9 in a varying number of colluders scenario (colluders return results 100% forged).

tem assigns extra work for disambiguation of the voting quorums. As the graph depicts, if the malicious group is small, as it is expected, the amount of work that has to be reassigned is minimal. Even in the worst case (where 50% of the participants are colluders), the amount of work performed is below the replication factor.

Although this technique improves the efficiency in terms of the amount of execution required, the late assignment of tasks can increase the overall time to perform the long running task. Nonetheless, standard assignment is also susceptible of late assignments, if no majority is gathered in the quorums.

Figure 5.3 depicts a comparison between the incremental assignment and the standard assignment, using replication factor 3, in a scenario where a number of faulty participants return results that always mismatch. The graphic shows that the amount of work converges for high percentages of faulty participants. Incremental replication is always more efficient in terms of amount of execution performed.

5.1.3 Replication using Overlapped Partitionings

Overlapped partitioning influences the way that the colluders introduce their bad results: it produces more points where collusion may happen but also where it may be detected; the size of each bad result is smaller, though. This happens because one task is replicated into more

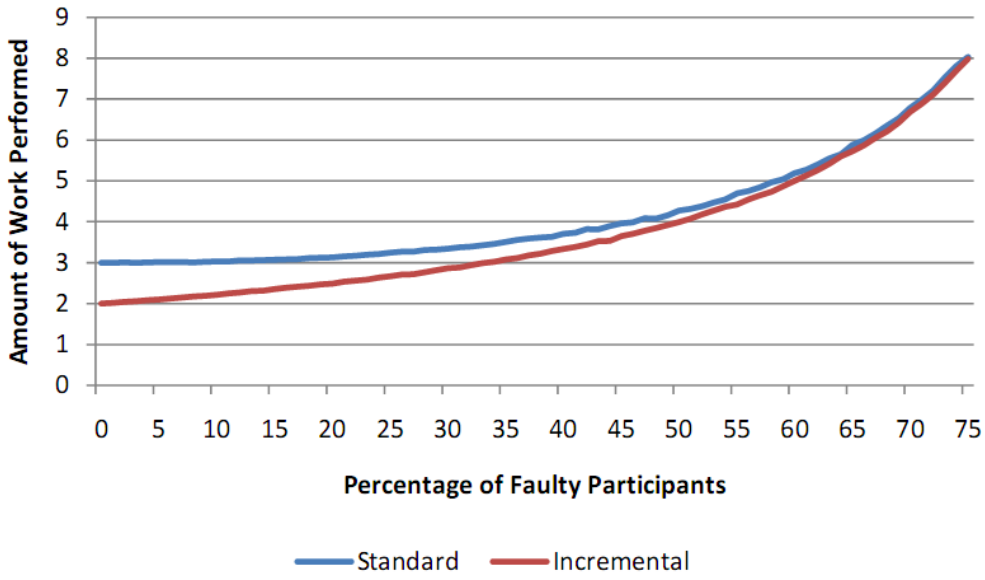

Figure 5.3: Amount of work performed using incremental and standard replication with replication factor 3 in a varying number of faulty participants scenario.

tasks than using standard partitioning; therefore there is a higher probability of redundant work being assigned to colluders; however they can only collude part of the task instead of the whole task as using standard partitioning.

The graphic in Figure 5.4 depicts that overlapped partitioning is as good as standard partitioning, in a scenario where the colluders are fully able to identify the common part and collude it, while executing the non-common part (in theory possible, but in practice harder to achieve as this may require global knowledge and impose heavier coordination and matching of information among the colluders). This is the worst case scenario, therefore overlapped partitionings can improve the reliability of the results depending on how smart the colluders are.

5.1.4 Replication using Meshed Partitionings

Meshed partitionings consider that some long running tasks can be divided in more than one dimension (e.g., ray-tracing image (2D), video transcoding (3D)). This n-dimensional partitionings provide many points of comparison, this information that is used to calculate the reputation of a result. Result reputations are used to disambiguate different results that were returned by redundant execution of the same task, lowering the amount of work that must be assigned for disambiguation. Furthermore, it does not use voting quorums, enabling the use of even

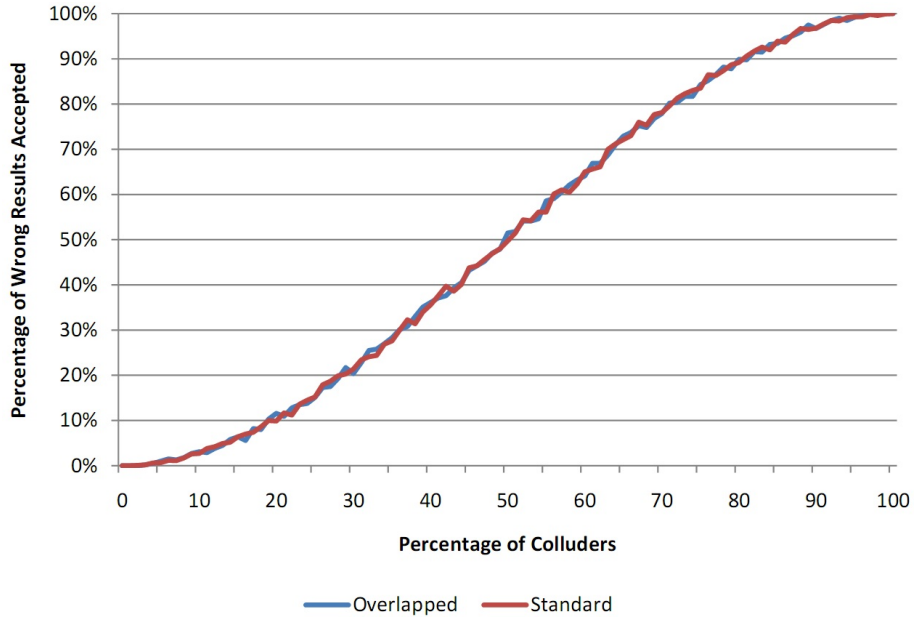

Figure 5.4: Replication using Standard Partitioning Vs. Replication using Overlapped Partitioning, using replication factor 3 (colluders return results 100% forged).

replication factors.

Figure 5.5 depicts the percentages of good results accepted, bad results accepted, and results that were not yet accepted and need rescheduling for disambiguation. In this simulation the colluders return results that are 100% forged, and bi-dimensional task partitioning is used (i.e., replication factor 2). As depicted, the amount of bad results accepted is zero for a considerable amount of colluders, the amount of work that has to be rescheduled grows along with the number of colluders until the percentage of colluders reaches 50%, though.

This technique is more efficient using replication factor 2 than standard replication using replication factor 3. Therefore, it has a more profitable ratio between the probability of an accepted result being incorrect and the overhead incurred by the technique. However, its implementation is a bit more complex than standard replication using voting quorums and it only fits some applications.

5.1.5 Replication and Random Sampling

As mentioned in Section 3, to base all the result verification mechanisms in information provided by non-trusted third parties might not be enough. Therefore, we consider the use of samples. In this section, we evaluate the use of sequential use of replication and sampling.

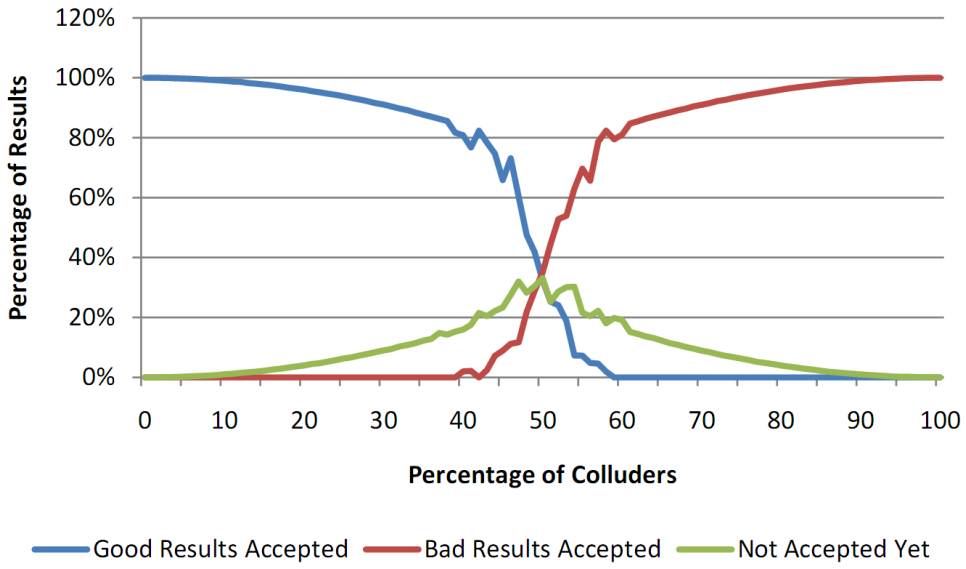

Figure 5.5: Replication bi-dimensional Meshed Partitionings before rescheduling of work, in a scenario where colluders return results 100% corrupted.

Using this technique, once a result has been chosen in the voting quorums it is submitted to a comparison with locally executed samples, chosen randomly. If the samples match, the result is accepted; if not, the non-matching results are dropped out of the voting quorums and recovery tasks are assigned and compared in the voting quorums again.

Figure 5.6 depicts the percentage of wrong results accepted using replication and sampling, using replication factor 3. Colluders return results 50% incorrect. It is clear that random sampling can have a very positive impact in the percentage of wrong results accepted, especially when the group of colluders is small. The reason this percentage of wrong results is so low is the following: once a wrong result has won the voting quorum, it means 2 or 3 results are to be compared with the samples; if it won the quorum with 2 equally wrong results and we are using 3 samples per task, it means we have 6 chances to hit a wrong sample result within the 50% portion of forged result.

This technique is, however, very wasteful, for it reschedules huge amounts of work and requires the local execution of a considerable amount of samples. Rescheduling incurs overhead in the amount of execution required, and the time to complete the long running task. The major problem is that this technique does not use all the information that it receives: voting quorums discard huge amounts of correct results.

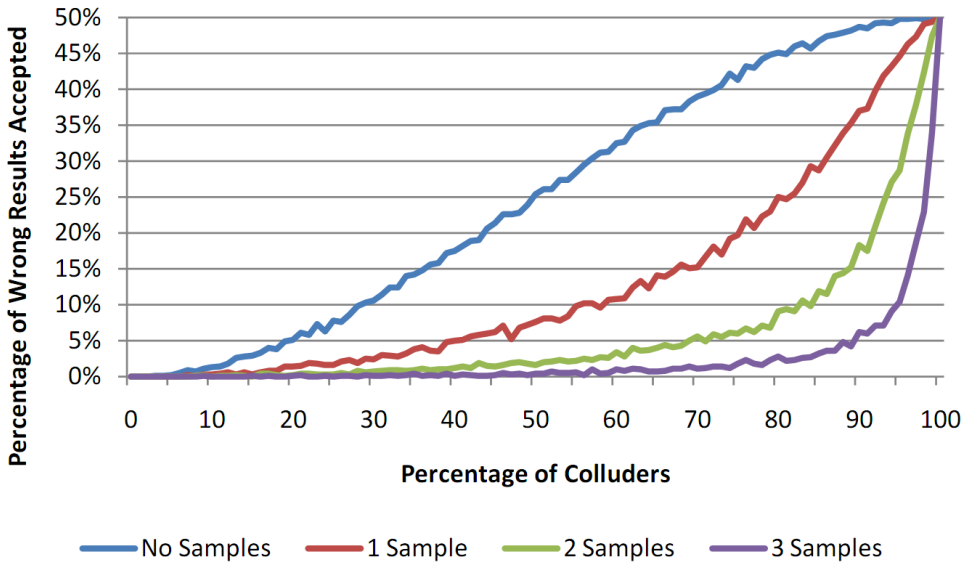

Figure 5.6: Replication and Random Sampling using replication factor 3 and different numbers of samples in scenarios with different amounts of colluders (colluders return results 50% forged).

5.1.6 Samplication

Samplication is a technique that combines sampling and replication without using voting quorums. It uses information from replication to decide where to choose samples, rather than selecting samples randomly. It chooses the samples within a replication mismatch area and discards the results that do not match the chosen sample. If there is no mismatch in replication, it uses random sampling.

As seen in Figure 5.7, this technique is very effective, it keeps the percentage of wrong results accepted very low, even for medium groups of colluders. Furthermore, this technique also works with even replication factors.

The graphic in Figure 5.8 depicts the average number of samples per gridlet (including the redundant ones). The amount of samples that are executed is very low, especially if the group of colluders is small. Furthermore, the number of samples is very well behaved and can be tightly controlled, making it very easy to find a compromise between the overhead and the reliability of the results.

Figure 5.9 depicts number of times the base work is executed. The required amount of work is almost the same as the replication factor requires, this means the rescheduling of work

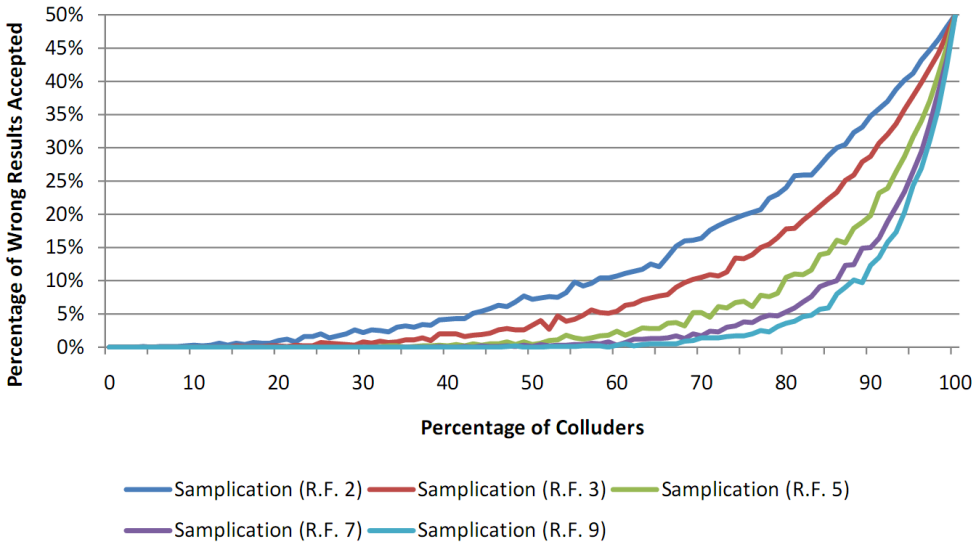

Figure 5.7: Result Verification - Samplication: percentage of wrong results accepted in a scenario where return results 50% corrupted.

is rare. It only makes a noticeable growth in scenarios with enormous groups of colluders.

Samplication is very well behaved technique due to its ability to disambiguate work using samples instead of rescheduling it. It can establish a good compromise between the reliability of the results and the incurred overhead incurred (locally by samples, and remotely by the replication factor).

Furthermore, the comparison of a trusted result with the results returned by the untrusted participants can accurately identify malicious isolated participants and members of a group of colluders. This technique also ensures that if in the redundant work that is assigned there is a correct result, that result will always be the one accepted, and the participant will receive credit for it. Therefore, this technique is an accurate source of information that can feed a reputation mechanism that influences the scheduler, improving the reliability of the results.

5.2 Checkpoint/Restart

Checkpoint/restart systems incur two different overheads: i) the extra time consumed to create a checkpoint; and ii) the size of the checkpoint. The extra time spent creating a checkpoint usually dictates the time interval between checkpoints (or the number of checkpoints per task), finding a compromise between the performance slowdown during fault-free execution and

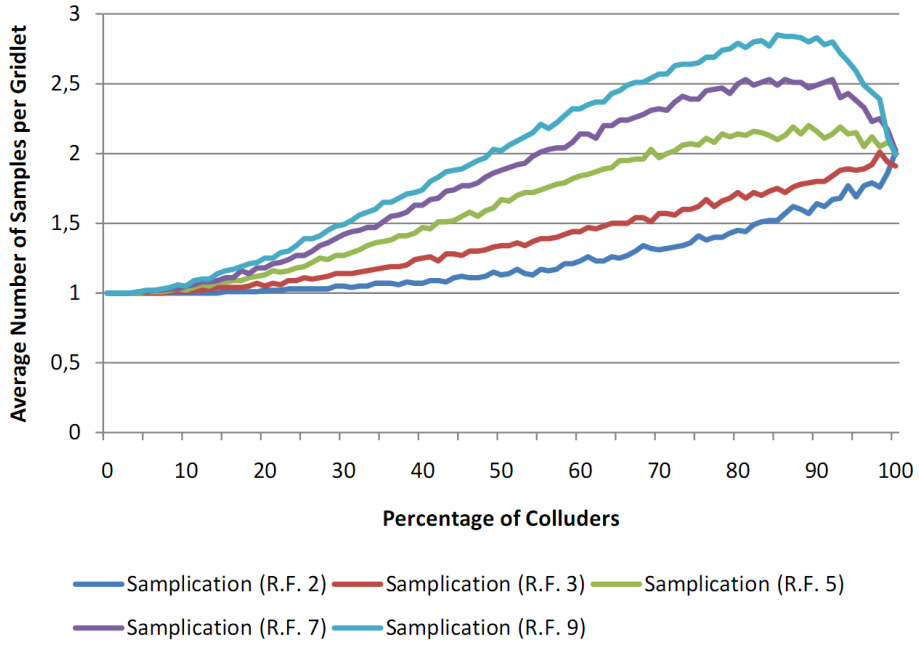

Figure 5.8: Result Verification - Samplication: average number of samples executed, using various replication factors (colluders return results 50% forged).

the loss of already performed execution due to the occurrence of a fault. The other aspect of concern is the size of the checkpoint data: this is crucial if the checkpoint is to be transmitted through the network (as is the case in our system).

5.2.1 Through a Virtual Machine's Running Image

The most relevant issue of this checkpoint/restart technique is the size of the checkpoint data to be transmitted. This is mitigated by the use of differential disk images, differential volatile state, compression, and use of optimized operating systems.

Differential disks are a feature supported by the main virtual machine implementations with specific disk image format files (e.g., QCOW2 (McLoughlin 2008)). The table in Figure 5.1 depicts the size of the checkpoint data to be transmitted attenuated with the use of differential disk images. The differential disk is an efficient representation of the modifications made to the virtual disk, these modifications are mostly the data written by the running application. Therefore, their size depends mostly on the data written by the running application.

Differencing the volatile state is an interesting feature that is not yet supported by the current virtual machine implementations. VirtualBox saves this volatile state in a complex, dy-

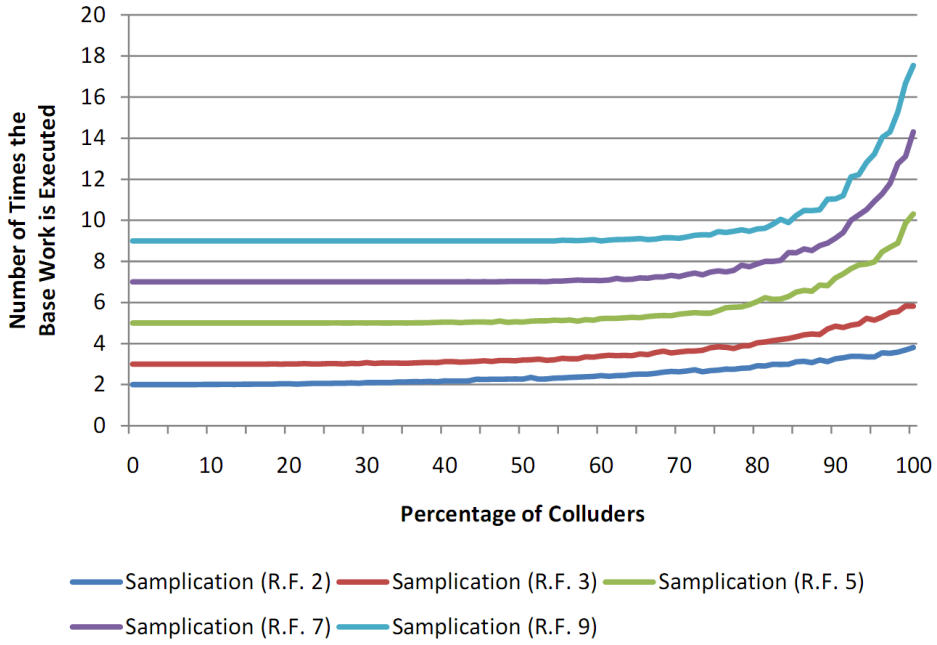

Figure 5.9: Result Verification - Samplication: number of times the base work is executed, considering the rescheduling (colluders return results 50% forged).

namic file (.SAV). Our effort to find redundancy between these SAV files was unsuccessful. We believe this is due to the complex data representation of these files. Nonetheless, differencing volatile states is still a valid technique to be further explored in other virtual machines that represent their volatile data in a less complex manner, or even using raw representation of the virtual machine's memory.

Compression enables a more efficient representation of the data in terms of size. We have applied compression to the previously referred checkpoints, the sizes of the checkpoints are depicted in Table 5.2. Compression reduces the size of checkpoints (SAV and differential VDI) in 50%.

The techniques used enable a more efficient representation of the checkpoint data using virtual machines. We have reduced the data to be transmitted from the unbearable 2.8 GB of running virtual machine state to 100 MB of transmittable checkpoint data. Ultimately, the size of the checkpoints is mostly composed by the application's disk written data and the memory loaded data. Therefore, checkpoint size will depend on the application.

Table 5.1: Checkpoint/restart through a virtual machine's running image: checkpoint data size using VirtualBox and Ubuntu Desktop 9.10.

			Data Size (KB)	
Base Image	Powered Off	disk image (.vdi)	2.651.169	2.768.998
	After Boot	disk image (differencial .vdi)	33	
		volatile state (.sav)	117.796	
Current Image A	Running Application A	disk image (differencial .vdi)	16.417	154.403
		volatile state (.sav)	137.986	
Current Image B	Running Application B	disk image (differencial .vdi)	23.585	209.597
		volatile state (.sav)	186.012	

Table 5.2: Checkpoint/restart through a virtual machine's running image: checkpoint data size using 7zip compression.

	Data Size (KB)	Compressed Data Size (KB)	Relation (%)
Current Image A	154.403	74.466	48,23
Current Image B	209.597	105.408	50,29

5.2.2 Through the Result Files

Our result checkpointing approach is based in capturing results during the execution on the participants and transmit those to the submitting participant incrementally. The incremental transmission overhead is negligible since these results would be transmitted at the end of execution anyway.

Checkpoint/restart mechanisms incur some permanent overhead during fault-free execution. The table in Figure 5.2.2 depicts the checkpointing time overhead incurred for 4 different rendering tasks using POV-Ray. The overhead of taking checkpoints using this technique is incurred by the several sequential re-launches of the application. As shown, the overhead depends on the duration of the task: taking 10 checkpoints during the execution of task 4 is almost negligible, while taking 1000 checkpoints along the execution of task 1 is a complete overkill.

Checkpoint/restart mechanisms start paying-off when faults occur. The table in Figure 5.2.2 depicts the pay-off of the checkpoint/restart mechanisms in the presence of a fault. It

Table 5.3: Checkpoint/restart through the result files: time overhead during fault-free execution.

	No Checkpoints	10 Checkpoints		100 Checkpoints		1000 Checkpoints	
	time (hh:mm:ss)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)
Task 1	00:00:17	00:00:22	29,41%	00:01:15	341,18%	00:10:15	3517,65%
Task 2	00:01:16	00:01:20	5,26%	00:02:08	68,42%	00:10:20	715,79%
Task 3	00:05:21	00:05:29	2,49%	00:06:38	23,99%	00:17:58	235,83%
Task 4	00:12:02	00:12:10	1,11%	00:13:01	8,17%	00:21:05	75,21%

Table 5.4: Checkpoint/restart through the result files: time overhead pay-off during faulty execution.

		No Checkpoints	10 Checkpoints		100 Checkpoints		1000 Checkpoints	
		time (hh:mm:ss)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)	time (hh:mm:ss)	overhead (%)
Fault at 25%	**Task 1**	00:00:21	00:00:24	13,88%	00:01:16	256,47%	00:10:16	2797,01%
	Task 2	00:01:35	00:01:28	-7,37%	00:02:09	36,08%	00:10:21	553,28%
	Task 3	00:06:41	00:06:02	-9,81%	00:06:42	0,18%	00:17:59	168,93%
	Task 4	00:15:02	00:13:23	-11,02%	00:13:09	-12,60%	00:21:06	40,31%
Fault at 50%	**Task 1**	00:00:26	00:00:24	-5,10%	00:01:16	197,06%	00:10:16	2314,18%
	Task 2	00:01:54	00:01:28	-22,81%	00:02:09	13,40%	00:10:21	444,40%
	Task 3	00:08:01	00:06:02	-24,84%	00:06:42	-16,52%	00:17:59	124,11%
	Task 4	00:18:03	00:13:23	-25,85%	00:13:09	-27,16%	00:21:06	16,92%
Fault at 75%	**Task 1**	00:00:30	00:00:24	-18,66%	00:01:16	154,62%	00:10:16	1969,29%
	Task 2	00:02:13	00:01:28	-33,83%	00:02:09	-2,80%	00:10:21	366,63%
	Task 3	00:09:22	00:06:02	-35,58%	00:06:42	-28,44%	00:17:59	92,09%
	Task 4	00:21:03	00:13:23	-36,45%	00:13:09	-37,57%	00:21:06	0,22%

reflects 3 different scenarios that depend on when the fault occurs, determining the loss of already performed execution and the cost of restarting it from the beginning on the approach without checkpoints. For simplicity, the times using checkpoint consider an extra portion of execution that corresponds to a checkpoint (this is the worst case scenario). Analysing this table we conclude the following facts:

- 1000 checkpoints never pay-off for the considered tasks;
- 1000 checkpoints almost pays-off for task 4 in the fault occurring at 75% of execution scenario and it is clear that it will pay-off for bigger tasks;
- 10 checkpoints pays-off for all tasks but the first one, in the scenario where the fault occurs

at 25% of the execution;

- 100 checkpoints is the clear median for the used tasks, the bigger tasks keep paying-off whereas the smaller ones do not;
- 100 checkpoints is better than 10 checkpoints for task 4 in all the fault occurring scenarios;
- the best pay-off is for task 4 using 100 checkpoints, in the case a fault occurs at 75% of the execution;

As a global conclusion of the efficiency of this checkpoint/restart technique: we can say it can be used very efficiently by finding of a compromise between the duration of the task and the number of checkpoints for each of the applications to be used.

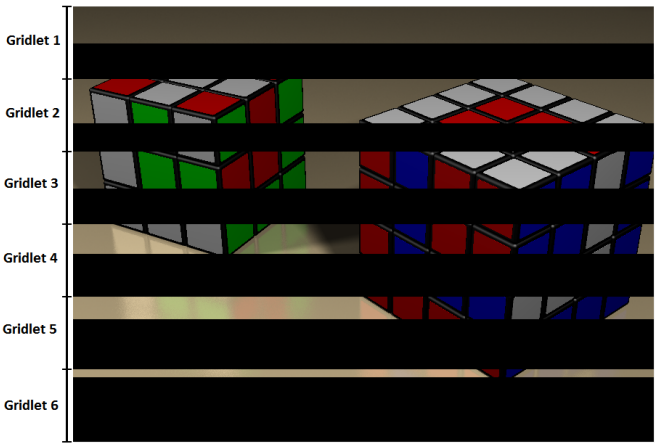

Figure 5.10: Checkpoint/restart through the result files: Previewing of a Ray-tracing result at execution time.

Furthermore, our technique enables the submitter to preview the results of the tasks that were submitted by him. This is a very pleasant functionality for the users, since they can see that the work they submitted is being executed. Although this seems like an extra functionality, it can have a very positive impact on the acceptance of these systems by the public. Figure 5.10[1] depicts the previewing of 6 independent POV-Ray tasks executing.

In this Chapter we have evaluated the tecniques that have been proposed earlier. The next Chapter concludes.

[1]The image depicted is based in a POV file subject to a Creative Commons - Lesser General Purpose License (CC-GNU LGPL) and can be found in the POV-Ray Object Collection at http://lib.povray.org/

6 Conclusions

In this work we have explored fault-tolerance and reliability mechanisms to improve the robustness of high performance computing systems in public environments.

In Chapter 1 we briefly introduced the context these systems. We described the challenges of these systems, the non-regular behaviour and faults that are expected and need to be handled for they can reduce the performance or even cause these systems to work inappropriately.

We studied the state of the art in Chapter 2. We analysed peer-to-peer and cycle-sharing for contextualization; and result verification and checkpoint/restart mechanisms as current solutions for the specific challenges that this work explores and its shortcomings.

Chapter 3 introduced GINGER overall architecture, defined the faults and non-regular behaviour to be supported, and introduced new ideas and techniques that could overcome the shortcomings of the current solutions. To attenuate the volatile nature of the participants we proposed checkpoint/restart approaches (through a virtual machine's running image and through the result files). To ensure that the results produced by these systems are reliable we proposed new result verification techniques based in redundant execution and comparison with locally calculated samples.

In Chapter 4 we described our implementation: a simulator that enables us to perform statistical analysis of result verification strategies defining environments with large populations, and a real deployment that proves that our techniques are feasible.

Finally, in Chapter 5 we presented an evaluation of the techniques proposed earlier. We evaluated our result verification strategies, as to their effectiveness (how high is the percentage of wrong results accepted) and to their efficiency (how much overhead it introduces). We evaluated the overhead of our checkpoint/restart approaches, as to our through the result files

approach we focused in the overhead introduced by taking checkpoints and to our approach using a virtual machine we focused on the size of the checkpoint data.

We have studied the shortcomings of current solutions and we have proposed, implemented and evaluated new approaches that improve the robustness high performance computing systems in public environments.

6.1 Future Work

The techniques proposed in this work can serve as the basis of future work in these systems.

The result verification techniques that we studied can feed a reputation mechanism that influences the scheduler. This can improve even more the reliability of the results, as we do not assign work to participants we believe are not reliable.

The checkpoint/restart mechanisms can be used not only for fault mitigation, but also to improve the overall performance of the system through migration. The correct evaluation of the present resources and prediction of the migration cost can determine if migration of a task is profitable, and if so its implementation can rely on the checkpoint/restart mechanisms we studied in this work to perform it.

Bibliography

Anderson, D. P. (2003). Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*.

Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Washington, DC, USA, pp. 4–10. IEEE Computer Society.

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

Anderson, T. E., D. E. Culler, D. A. Patterson, , & the NOW team (1995). A case for now (networks of workstations). *IEEE Micro 15*, 54–64.

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

Baldoni, R., J.-M. Hélary, A. Mostefaoui, & M. Raynal (1995). On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems. Research Report RR-2569, INRIA.

Barkai, D. (2001). Technologies for sharing and collaborating on the net. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing*, Washington, DC, USA, pp. 13. IEEE Computer Society.

BitTorrent (2003). Free, open source file-sharing application effective for distributing very large software and media files. In `http://www.bittorrent.com/`.

Burns, R. C., A. C. Burns, & D. D. E. Long (1997). A linear time, constant space differencing algorithm. In *In Performance, Computing, and Communication Conference (IPCCC*, pp. 5–7. IEEE International.

Cezário, J. & A. Sztajnberg (2008, July). Introdução de um mecanismo de checkpointing e migração em uma infra-estrutura para aplicações distribuídas. In *V Workshop de Sistemas Operacionais (WSO'2008)*.

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

Clarke, I., O. Sandberg, B. Wiley, & T. W. Hong (2001). Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science 2009*, 46–66.

Costa, L. B., L. Feitosa, E. Araujo, G. Mendes, R. Coelho, W. Cirne, & D. Fireman (2004). 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*.

distributed.net (1997). Distributed.net: Node zero. In `http://distributed.net/`.

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

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

Elnozahy, E. N. M., L. Alvisi, Y.-M. Wang, & D. B. Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*(3), 375–408.

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

Foster, I. & C. Kesselman (1996). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications 11*, 115–128.

GIMPS (2010). Great internet mersenne prime search. In `http://mersenne.org`.

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

KaZaA (2000). Download music, free music downloads. In http://www.kazaa.com/.

Larson, S. M., C. D. Snow, M. Shirts, & V. S. Pande (2009, Jan). Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. Technical Report arXiv:0901.0866.

Lawall, J. L. & G. Muller (2000). Efficient incremental checkpointing of java programs. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, Washington, DC, USA, pp. 61–70. IEEE Computer Society.

Litzkow, M., M. Livny, & M. Mutka (1988, June). Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104–111.

Litzkow, M., T. Tannenbaum, J. Basney, & M. Livny (1997, April). 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.

Lo, V., D. Zappala, D. Zhou, Y. Liu, & S. Zhao (2004). 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*, pp. 227–236.

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

Maymounkov, P. & D. Mazières (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, London, UK, pp. 53–65. Springer-Verlag.

McLoughlin, M. (2008). The qcow2 image format. In http://people.gnome.org/~markmc/qcow-image-format.html.

Molnar, D. (2000). The seti@home problem. In *ACM Crossroads: The ACM Student Magazine*.

Mutka, M. W. & M. Livny (1988). 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*, pp. 529–544.

Napster (1999). Stream music, download mp3s, top songs, buy music - napster. In http://free.napster.com/.

Plank, J. S., M. Beck, G. Kingsley, & K. Li (1995, January). Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pp. 213–223.

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

QEMU (2010). Qemu is a generic and open source machine emulator and virtualizer. In http://wiki.qemu.org/.

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

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

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

Shirky, C. (2002). Clay shirky's writings about the internet. In http://www.shirky.com/writings/napster_speech2.html.

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

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). 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*, New York, NY, USA, pp. 149–160. ACM.

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

Veiga, L., R. Rodrigues, & P. Ferreira (2007, May). Gigi: An ocean of gridlets on a 'grid-for-the-masses´. In *IEEE International Symposium on Cluster Computing and the Grid - CCGrid 2007 (PMGC-Workshop on Programming Models for the Grid)*. IEEE Press.

VirtualBox (2010). An x86 virtualization software package developed by sun microsystems. In `http://www.virtualbox.org/`.

Zhao, S., V. Lo, & C. GauthierDickey (2005). 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*, pp. 31–38.

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