BARRAGE - Data Storage in BAR Model

Xavier Vilaça
xvilaca@gsd.inesc-id.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

January 6, 2011

Abstract

Peer-to-peer networks have emerged as a relevant architecture for executing scientific computations. However, this kind of networks poses several problems. This thesis addresses the problem of data storage in a peer-to-peer architecture. We consider several distinct behaviors of the participating nodes. To deal with this variety, we use the BAR model, that comprises byzantine, rational, and altruist peers. In this report, we provide a survey on peer-to-peer storage solutions and on different approaches to deal with byzantine and rational peers. We also propose a new sub-system for data storage that tolerates byzantine behavior, by assuming an upper bound on the number of these nodes. It also provides incentives to rational peers to behave as desired, by constantly monitoring their actions, rewarding good behavior, and applying appropriate penalties to misbehaved nodes.

1 Introduction

Scientific computation involves the execution of CPU intensive operations on very large datasets. Many relevant problems in this area have scalability requirements that cannot be satisfied by centralized architectures, even by those custom built for that particular purpose, such as supercomputers. Therefore, these programs need to be executed in parallel on multiple machines.

To achieve this parallelization, it is required to partition the work into multiple tasks. Several programming models have been proposed to address this need. One of the simplest is the *bag-of-tasks* model, where the application is decomposed on a number of (worker) tasks that are managed by a single master task. A well known example is the BOINC system[4]. This model is only appropriate to *embarrassingly parallel* applications, where tasks are not required to communicate among each other during their execution (i.e., tasks only communicate by reading the input values when they begin and by producing output values when they complete). The *Map-reduce*[13] model emerged as a refinement of the previous paradigm. It also follows a master-slave model, but tasks are further classified as map, executed by *mappers*, or reduce,

executed by *reducers*. The work is split into map tasks that produce a set of intermediate results. Later, these are consumed by reduce tasks that generate the final results. Map-reduce has become very useful for making simple computations, like sorting and word counting, on huge files, providing a satisfactory response to the need for processing large amounts of raw data, like web documents and logs. The programming model of Map-reduce is easy to grasp by programmers and allows a good partitioning of work without much effort. In this thesis we are mainly concerned with providing support for computations based on map-reduce.

In what concerns the support for parallel distributed computations, there are many well known examples of different distributed architectures that have been proposed for that purpose, including clusters, enterprise grids, cloud computing, and peer-to-peer(P2P) networks. P2P grids are important due to the fact that they allow to use the huge amount of storage space and CPU capacity that can be made available by most computers of the Internet. This is of particular interest to organizations that cannot afford to build or rent expensive infra-structures.

Unfortunately, P2P grid computing also has its own drawbacks. Since nodes are not under the control of a unique centralized organization, they can join and leave the network at any moment, for arbitrary reasons. They can also assume a malicious behavior, by trying to attack the system. Even if they are not malicious, they may not necessarily present an altruistic behavior. Selfish nodes usually strive to minimize their contribution for the system while maximizing their profit. Systems that strive to tolerate both rational and malicious peers are modeled by the Byzantine Altruist Rational (BAR) model[3]. In this thesis we are interested in designing solutions for P2P grid computing under the BAR model, that address some of the challenges enumerated above.

One of the key components of such architecture is a distributed P2P storage subsystem that supports data exchange among mappers and reducers. The thesis will focus on exactly this topic. The idea is to build a storage sub-system that allows to maintain intermediate results in volunteer nodes, named *storers*, that are neither mappers nor reducers. Mappers can use storers to save their results to be later consumed by reducers, decoupling those identities.

The rest of this document is organized as follows. Section 2 specifies the goals for this work and gives an overview of the expected contributions. Section 3 describes previous work related to the problems that were introduced. Section 4 gives a description of the proposed architecture. In Section 5, a plan for evaluating the proposed solution is presented. Section 6 describes the schedule of future activities, and Section 7 concludes the report.

2 Goals

This work addresses the problem of designing and implementing a distributed storage sub-system for P2P grid computing that supports the exchange of data performed by different participants. Given that P2P networks are subject to the existence of malicious, selfish and altruist participants, the storage sub-system will consider the presence of a mix of byzantine, altruist, and rational nodes, as captured by the BAR system

model[3]. Due to this, we will call our system BAR stoRAGE or simply, BARRAGE.

To enforce rational nodes to provide service, we will need to design monitoring mechanisms that: i) are able to certify that correct nodes did, in fact, provide the service that they volunteered to provide; ii) can detect faulty nodes that do not provide the required service and produce proofs of misbehavior and; iii) provide proofs of correct behavior. Therefore, the thesis will:

> *Goals:* Design and validate a set of protocols to be executed among storage, producers, consumers and monitor nodes that certify that rational nodes provide the expected storage service, and tolerate different possible behaviors of byzantine participants.

For scalability reasons, the monitoring of storage nodes will not be performed by a centralized trusted node. Instead, monitoring will be distributed, and performed by volunteer nodes (that may also provide storage service or act only as monitors). As a result, one cannot trust on the output of a single monitor to assess the behavior of storage nodes, since the monitor may be malicious. Even if the monitor is not malicious, if the proper protocols are not put into place, the rational behavior may cause the monitor to skip some of its duties. Therefore, to develop our solution we will need to seek inspiration in byzantine fault-tolerant monitoring mechanisms, such as Fireflies[24].

The expected results from this project are the following:

> *Expected results:* The work will produce i) a specification of the protocols to be executed among producers, storers, consumers, and monitors; ii) an implementation of the BARRAGE system; iii) a theoretical analysis of the resulting system.

3 Related Work

This section provides an overview of the related work on storing data in large P2P systems considering the possible different behaviors of peers. A special emphasis is given to the BAR model and its main advantages over other alternative models when it comes to portray the behavior of nodes. We also address the concept of ensuring data storage on remote nodes by monitoring them and giving incentives.

Firstly, we introduce the concept of P2P storage. Then, we describe the fault model. Finally, we survey related work on byzantine and rational fault tolerance, reputation management, BAR protocols, and monitoring data storage.

3.1 P2P Storage

P2P systems are designed to support direct interactions among peers, instead of relying on central servers. This makes these architectures particularly appealing to develop distributed storage systems, given that they have the potential to provide *high availability*, *load balancing*, and *scalability*. Since peers can communicate directly and data can be stored by different peers, located in different geographical regions, such a system can be designed to tolerate failures of individual nodes, or even natural

catastrophes. Furthermore, since different data is stored in different and multiple peers, it is less likely that a single node or link becomes a bottleneck of the entire system. Finally, since the architecture is able to use the resources provided by each participant, the amount of available resources increases with the number of nodes, making the architecture scalable.

In this section, we outline the most important aspects that make this architecture interesting for supporting data storage in grid computing. We also highlight the major challenges that must be faced, when designing such systems.

3.1.1 P2P Storage Systems

Many different distributed P2P storage systems have been proposed in the literature and/or have been used in real-life systems. It is possible to classify these systems according to the type of functionality they provide. Naturally, systems built for different purposes have different use patterns and, therefore, rely on different techniques. There are three main types of P2P systems that support data storage, namely, *Generic File Systems*, *Content Sharing Systems*, and *Data Storage Systems*:

Generic File Systems This is the most broad class of storage systems. It aims at providing, in a distributed environment, an abstraction that is as close as possible to that of a local file system. Data is divided in files, that are organized in a hierarchy of directories. Any user can access, create, modify, or delete files, and access, create or delete directories. These systems have to deal with concurrent manipulation of files and directories, and address any conflicts that may arise from this concurrency. Also, data must be persisted until it is explicitly deleted. Some examples of these are CFS[12] and Ivy[34].

Content Sharing Systems These systems are optimized for sharing, searching, and publishing digital media in the Internet. Usually, each content object is owned by a restricted group of hosts. Unlike the *Generic File Systems*, only the owners may perform changes in those objects; any other host is only capable of reading the data. Thus, conflicts do not arise so often, since the number of owners is limited and they may synchronize their operations in order to avoid concurrent modifications on objects. These are probably the most popular and widely used P2P systems with storage features. Well known examples are Bittorrent[8], Napster[1], and Kazaa[30].

Data Storage Systems These systems aim at preserving data on the network. In the previous two types of P2P systems, focus was given to the distribution of the information through all the participants, and management of the operations over the data. Data Storage systems are more concerned with data durability and availability. Normally, there are one or more participants that produce the data and store it on the network, relying on data redundancy mechanisms to ensure the availability and durability of the information. Systems like Oceanstore[26] allow different users to manipulate the data concurrently and deal with conflicts.

[1] www.napster.com

Others, such as PAST[15], Freenet[7], and backup systems[16, 5, 17], do not deal with concurrent writes.

The BARRAGE sub-system belongs to the class of Data Storage Systems.

3.1.2 Design Issues

P2P storage networks present several problems that must be addressed when designing new systems. In the following, we highlight some of them:

Churn Churn[20] is the phenomena that consist in having multiple peers leaving and joining the P2P network at a fast pace. This increases the signaling traffic in the network and may lead to a poor performance of the system or even service unavailability due to convergence problems. Naturally, systems that require a larger amount of control information to maintain their operation are less scalable and more susceptible to churn.

Data Persistence As the name implies, some storage systems are concerned with ensuring the durability and availability of the data stored in the system. Given that in a P2P system nodes are likely to eventually fail or leave the system, P2P storage systems implement data redundancy techniques to achieve some measure of data persistence.

Content Location Given that P2P systems are dynamic and may be subject to churn, it may become difficult to keep track of the location of the replicas of a given data item. Resource location may be purely reactive (by executing an expensive search algorithm when data is needed), or pro-active, by maintaining routing information that supports the location procedure. Naturally, there is a trade-off between performance and maintenance cost.

Failures At any given instant, a peer can fail due to arbitrary reasons. Power loss, hardware failure, software bugs, malicious users, are some of the reasons that may cause a node to violate the expected behavior. Therefore, a P2P system must be prepared to tolerate the faulty-behavior of a subset of its peers, ensuring that data integrity is not compromised by attacks performed by malicious nodes.

Rational Behavior Many P2P systems rely on nodes that offer their resources voluntarily. These nodes may not necessarily be altruistic, i.e., willing to offer more resources than they strictly have to, in order to extract their desired benefit from the system. A node that has no explicit intent to attack a system but that is able to control its behavior to maximize a local utility function is denoted a rational node.

3.1.3 Security Issues

In the following, we enumerate some of the main attacks that may be performed by malicious nodes and that can compromise the operation of a P2P storage system.

Denial of Service A serious threat to any system connected to a public network is the possibility of a malicious peer (or, most likely, a group of colluding peers) to flood the network with messages or overload another peer with requests. This prevents the system from operating correctly, hence denying the service.

Sybil Attack A sybil attack occurs when a group of malicious nodes (eventually one) presents itself to the network with more identities than the number of members of the group. If the malicious nodes are powerful enough, they can emulate as many false nodes as required to take full control of the system.

White-washing In the same way as in the sybil attack, a node can constantly change its identity to avoid penances due to misbehavior.

Eclipse Attack This is a special attack that strives to divide the network into two or more sub-networks by placing faulty nodes, such that all paths between any pair of sub-networks always contains a malicious node. This allows an attacker to isolate one or more nodes and control its participation in the system, by *eclipsing* them.

Deception Deception is a technique where a malicious node provides false or distorted information to other nodes in order to induce them to engage in a behavior that is prejudicial to the system goals. For instance, in a P2P storage system, a node may claim that itself or other peers have data that they actually do not store, or claim that some correct nodes are faulty or do not own the data they are supposed to store.

3.1.4 Rational Behavior Issues

In opposition to a malicious node, a rational node will not attack a system with the only purpose of making it unusable. Instead, a rational node will only deviate from the intended behavior if this allows him to maximize a local utility function. For instance, a rational node may avoid storing data from other nodes if the protocols allow it do so while still getting its own data stored by other nodes. Still, this may be enough to prevent the system to operate as intended, or even fail as a whole if a large fraction of participants present a rational behavior. Therefore, it is important to take measures to mitigate the negative effects of this kind of behavior.

Free-Riding The problem of free-riding in the context of P2P has been discussed in [1]. Free-riders will exploit the altruism of other nodes to take advantage of the system without reciprocating in a reasonable manner, i.e., providing at least as much resources as they consume. Experience with several practical systems[38, 30] has shown that this problem can ultimately prevent the system from working. This leads to a situation called *Tragedy of the Commons*[23], where every node is expecting others to provide services, but that never happens and the system stops working.

Alternative Behavior Most protocols provide different options for each step during an interaction. It is possible for a rational peer to take advantage of this and choose

a particular option that minimizes its cost or allows it to gain unfair advantages, instead of making the adequate choice. For instance, a rational node may omit messages to save network resources.

Collusion This problem occurs both with malicious and rational nodes. Collusion happens when several nodes coordinate their actions to defeat the mechanisms in place that aim at preventing or detecting incorrect behavior. Naturally, dependable mechanisms that are able to deal with collusion are more expensive and complex.

3.1.5 Relevant Aspects of P2P Storage Systems

We now address some relevant aspects of P2P storage networks namely, we address the following topics: *levels of decentralization* (the amount of tasks that are performed by centralized components), *network structure* (how peers self-organize), and *redundancy* (how data redundancy is managed).

3.1.6 Levels of Decentralization

All P2P systems leverage on the existence of multiple peers to decentralize several functions. However, decentralized control is more complex and often more costly than centralized solutions. Therefore, practical systems sometimes use a mix of both approaches. As a result, we can identify three main types of P2P architectures: *Fully Distributed*, *Partially Centralized* and *Hierarchical*.

Fully Distributed Here, the network does not depend on the services provided by any central entity. Relationships between entities are symmetric and any peer can start a communication channel with any other participant. This is the only architecture that can totally avoid single points of failure, censorship, and bottlenecks[38].

Partially Centralized In this type of architecture, central entities are used to perform specific functions like monitoring membership[3, 29, 28] and routing[2].

Hierarchical This is a hybrid approach between the previous two architectures. Instead of relying on central servers, some peers are elected supernodes and assume centralized roles, for a fraction of other (regular) peers[30].

3.1.7 Network Structure

In a P2P system, nodes organize themselves in an overlay network. The topology of this overlay is of crucial importance to certain aspects such as content location and membership management. It influences the scalability, availability, and performance of the system. Network overlays can be classified as *Structured* or *Unstructured*.

Unstructured - In an unstructured overlay, each node connects to a random subset of peers[38, 30]. Therefore, their placement in the network is not determined

[2] www.napster.com

a priori by their identifier. A fundamental advantage of this topology is that it has a low maintenance cost and is more resilient to churn. However, in this sort of overlay, resource location has to be performed by (blind) mechanisms such as flooding or random walks, given that the overlay provides little or no support for deterministic routing. This can be a major impairment to the scalability and performance of the resource location functions.

Structured - In a structured overlay, nodes logically position themselves in the overlay according to some deterministic strategy. This makes possible to perform deterministic routing at the overlay level, and implement Distributed Hash Tables(DHT), that maps content identifiers to nodes. Resource location can be performed with logarithmic complexity. Still, this requires greater maintenance costs and storage space in each node. Some solutions that fall into this class are PAST[15], Tapestry[42] and Chord[40].

Concerning membership, there are two important aspects to consider. First, the dimension of the system view, this is the number of nodes that each peer knows about. The second question is how that information is maintained.

Full vs Partial Memberships : Peers can keep a full view of the system by storing local information about all the nodes of the network[24]. This requires greater maintenance costs but it is more resilient to malicious behavior, since there is no need for searching. The other alternative is to keep a partial view of the system[15, 8]. This approach requires lower maintenance costs and is more scalable. However, there is the need for searching nodes for storing the data which requires several steps. Hence, the probability of encountering malicious nodes along the search route is higher. It is important to notice that some solutions[21] allow a constant number of steps without keeping a full membership.

Static vs Dynamic Membership : Not all P2P protocols strive to maintain an updated membership. Some solutions simply assume that nodes do not leave the network unless when they fail[3, 29]. This assumption might be reasonable for an organizationally owned network. However, it is not suitable for a P2P network deployed over the Internet. Hence, a practical system must implement mechanisms to keep membership information up-to-date[28, 24].

3.1.8 Data Redundancy

To ensure both availability and durability of data, mechanisms to provide data redundancy must be in place. There are two mechanisms for achieving this: Replication and Erasure-codes.

- In replication, data is copied to *r* replicas[15, 7, 6]. One advantage of this technique is that any replica that has the entire data can be used to recover it. So, with a replication factor of *r*, the system tolerates $r - 1$ failures. The main disadvantage of replication is that the global capacity of the system is reduced by a factor of r[6].

- To overcome the space inefficiency of replication, a more sophisticated technique was developed. Erasure-codes use redundancy in a different manner[2, 28]. Data is split into s pieces. Then a transformation is applied to the data, producing $s+r$ blocks[37]. Any subset of these blocks, of cardinality s, can be used to obtain the initial information. This strategy is more efficient concerning space usage but it requires more replicas to store and recover data[6] and makes updates more costly.

3.2 Fault Model

A P2P overlay is a network composed of processing entities, named peers or nodes, and communication links between them. These links can be *reliable* if it is guaranteed that every sent message is delivered, or *unreliable* otherwise. Regarding time of operations and message latency, systems can be *synchronous* if there is an upper bound for those values. If it is not possible to make timing assumptions about operations, the system is *asynchronous*.

3.2.1 Fault Prevention and Tolerance

A fault occurs when a node deviates from the specified behavior, either due to software bugs, malicious, rational intentions, or any other kind of software or hardware errors. Eventually, a fault may lead to a failure when the effect of the fault becomes externally visible.

There are two main complementary techniques that may be used to address the problem of faults in systems namely, *fault prevention* and *fault tolerance*.

Fault Prevention Techniques for preventing faults increase the dependability of the system by reducing the likelihood of fault occurrence. Good programming practices can prevent software error and avoid vulnerabilities that may lead to security attacks. If security is taken into account, it may be possible to prevent malicious nodes from disclosing or disrupting information, and from forging identities. Incentive mechanisms can provide enough reasons for rational nodes to respect the specified protocols, if this is the alternative that maximizes their profit.

Fault Tolerance Fault tolerance consists in adding mechanisms that allow the system to continue to provide the intended service despite the occurrence of faults. Fault tolerance may be achieved by detecting the faults and applying corrective measures, or simply by masking the fault using redundancy. In the context of P2P storage, it aims at avoiding data loss or corruption and the disruption of the network.

This thesis focuses, specially, on applying fault tolerance techniques. In the current literature, the following fault models characterize the different types of faults that may be tolerated: *Crash-stop Fault Model*, *Crash-recovery Fault Model*, *Byzantine Fault Model*, *Rational Nodes Fault Model*, and *BAR Fault Model*.

3.2.2 Crash-stop and Crash-recovery Fault Models

In the crash-stop model, a faulty node executes the specified behavior until it crashes, i.e., stops executing operations. Therefore, a faulty node never executes erroneous steps or produces incorrect results. A node is said to be correct if it never crashes.

On the other hand, in the crash-recovery model, a crashed process may later recover and resume operation. In this case, a correct process may crash and then recover multiple times, but eventually stops crashing.

In practice, correct nodes are not required to never crash (or stop crashing forever). It is enough that they remain active until the service they are supposed to provide is concluded.

In a P2P storage system, the disconnection of a peer from the network can be modeled as a failure. However, the peer may be considered correct if it successfully provided the expected service before abandoning the network, for instance if it is able to store and maintain the data available for a predefined period of time or until some event happens.

3.2.3 Byzantine Fault Model

The Byzantine Fault Tolerance model classifies nodes as either correct or byzantine. The word byzantine, in this context, was firstly used by Lamport et al., in [27]. He introduced the issue about malicious nodes by making an analogy. The problem is formulated as a group of byzantine generals who are planning an attack. They want to agree about a plan of action. But some of the generals might be traitors and suggest wrong decisions. So it is desirable that all loyal generals reach the same decision and that it is not influenced by a small group of traitors. In addition, the proposed value of a loyal general must be adopted as his choice by all the other loyal generals, even if it is not the final decision. Although, in this abstraction, we are only concerned with malicious peers, the fault model defines byzantine as all the nodes that present an arbitrary behavior. That includes both malicious actions and failures by crash or by leaving the network. A Byzantine Fault Tolerant (BFT) system does not fail even if a small fraction of the participants is byzantine.

3.2.4 Rational Fault Model

This model characterizes another class of nodes that are neither byzantine or correct. Nodes are classified as rational if their main goal is to maximize their profit. Assuming that every system provides some source of benefit to its users, rational peers strive to minimize their costs and maximize the benefit, even if that requires harming other nodes or cheating. Since it is not possible to avoid this behavior, the goal of this model is to tolerate it. This is done by conciliating the interest of rational nodes with the common good. This model does not distinguish rational behavior from altruistic behavior.

One important problem that occurs in the interaction among rational peers, that needs to be considered in this work, is the problem of *Fair Exchange*. This occurs

when, in a interaction between two entities, each one has information that is valuable to the other, and they want to exchange it. The problem is that one might try to obtain its information without providing information in exchange. In Section 3.3.2, we will discuss some solutions that address this problem.

3.2.5 BAR Fault Model

BAR stands for Byzantine, Altruist, and Rational. The model combines the previous models, in what concerns the behavior of nodes. Byzantine are incorrect nodes just like we have described in the Section 3.2.3.

Rational peers are characterized by a utility function that represents the expected profit of peers and rational peers try to maximize. This utility function captures what these nodes seek to achieve so we can model our protocols to conciliate that function with the common good. On the other hand, byzantine nodes may also be rational, in the sense that they may also adopt a well defined strategy that can be modeled by an unknown utility function. To model a rational behavior, it is necessary to consider what are the *benefits* from using the system and what are the *costs*. Then, it should be ensured that the benefit of deviating from the protocols is at least the same as not doing it.

Altruist nodes contribute to the system in a volunteer manner. Thus, they follow the protocol with or without incentives. The existence of this last type of peers is not mandatory. In some cases, making assumptions about the fraction of altruist nodes makes the system more practical. But a BAR Fault Tolerant system should be robust even when all nodes are either byzantine or rational.

One last important remark about this model is that only byzantine nodes are considered faulty. All rational and altruist peers are classified as correct because it is of their best interest that the system provides certain services. With byzantine nodes, that is not guaranteed.

3.3 Building Blocks for P2P Storage

There are many different approaches to the problems that emerge from building P2P storage systems. In this section, we describe the main building blocks for systems that tolerate byzantine and rational nodes.

3.3.1 Byzantine Protocols

In this section we introduce the main approaches to provide data storage in the presence of byzantine nodes, namely, *quorum-based*, *replica-based*, and *hybrid* approaches.

Quorum-based: A possible way of storing data and tolerating a certain number of byzantine failures is to use the notion of quorums[33]. A quorum is a subset of peers with which the client communicates directly. All read and write operations are applied to a given quorum, guaranteeing that any two operations intersect in a sufficient number of non-faulty nodes, such that correct values may be returned. Quorum approaches also provide a form of load balancing, since it is

not necessary to include all nodes in all operations. On the other hand, many quorum based protocols perform poorly when there is heavy contention between multiple writers.

Replica-based: The replica based approach[2, 26] consists on the use of replicas that communicate with each other to reach an agreement about the ordering of requests and the values to be stored. These protocols require the exchange of a large amount of messages, and present a low scalability capacity with the number of replicas.

Hybrid Approaches: As the name implies, these approaches combine the two previous schemes[9]. Here, a quorum of peers is used to obtain a majority of confirmations for each operation. In the presence of contention between writers, a replica-based approach is used to determine the final order of the conflicting operations.

Naturally, safe operations cannot be achieved unless the number of faulty nodes is bounded. In [27], it was proved that it is only possible to solve the byzantine generals problem with at least $3m + 1$ generals, for m byzantine generals, if messages are not signed(the sender can forge messages). It is also shown that, with signed messages, it is possible to lower this bound to $2m + 1$ generals. These bounds are only applicable if the system is synchronous and communication channels are reliable.

3.3.2 Incentive-based Protocols

Incentive-based protocols implement mechanisms that reward nodes for offering the intended service. The current state of art includes systems based on the following mechanisms: *direct reciprocity*, and *reputation*.

Direct Reciprocity: In direct reciprocity-based protocols, instead of being paid for a service, each node always provides a service in exchange for the service it receives. When storing data, nodes have to store locally the same amount as they want to store on other peers. This is frequently used in backup systems[10, 3, 16].

These schemes operate best if they can solve the *fair exchange* problem. In [18], its proved that this problem can only be solved with the help of a trusted third party. In [19], the authors propose an algorithm where the third party is only required to participate in the protocol when disputes arise.

Reputation: In these protocols, the system keeps track on the quality of services that a node has provided to the system using a reputation value. When a node provides service, its reputation increases. If the node fails to provide service it was supposed to provide, its reputation decreases. Therefore, the strategy of a rational peer is to maximize its reputation. Several solutions fall into this class[8, 22, 41]. Reputation protocols can be divided in three classes:

Local Reputation Systems: Each node keeps the reputation information of other nodes and updates it at each interaction[8].

Voting Reputation Systems : This scheme makes use of the local reputation information from different nodes in order to get a more accurate estimate of the real contribution of a given target node to the system. Whenever there is the need to determine the reputation of a node, a voting process takes place[41].

Transaction-based Systems : Nodes use proofs that a certain service was provided, also known as certificates, to assert their reputation[22].

3.3.3 BAR Protocols

Protocols devised for the BAR model assume that rational peers strive to minimize costs, such as bandwidth, storage space, communication, and amount of computation. However, it is also assumed that these nodes benefit from the services provided by the protocols. Hence, the expected behavior of each rational peer (strategy) consists on maximizing the benefits and minimizing the costs. With regard to protocols, this may imply sending the shortest messages on each step or avoid sending any message at all, sparing as much storage space as possible, and minimizing the amount of computations performed.

Existing implementations that adopt the BAR model strive to guarantee that, for each step of the protocol, the best strategy for a rational peer is to perform as expected. For this purpose, the concept of (exact) *Nash equilibrium* from Game Theory [35] is applied. In a Nash-equilibrium, a rational peer does not benefit more from changing its strategy. Most of BAR implementations[3, 29] aim at guaranteeing that each step of the protocol is an exact Nash-equilibrium, and provide informal proofs that this property holds.

A more flexible approach is taken in [28], by using the concept of (approximate) ϵ-Nash equilibrium. Here, it is possible that alternative strategies provide a greater benefit. The ϵ consists on an upper bound to the factor by which the expected benefit increases by changing strategy. In an ϵ-Nash equilibrium, rational peers only change their strategy if the expected raise of the benefits is greater than ϵ.

All the existing solutions in the literature adopt several measures for ensuring an exact or approximate equilibrium, as described below:

- Messages have a strict format and are digitally signed. A signed message that is wrongly formated, or sent in a wrong protocol step, is a *proof of misbehavior*(POM).
- The number of available choices is minimized, in order to reduce the opportunities for the rational node to select a behavior that is not beneficial to the system. For instance, non-deterministic choices should be avoided, as the rational nodes may select the most favorable choice instead of using a random variable. This can be achieved using verifiable pseudo non-determinism, that achieves good probabilistic distribution while preventing nodes from manipulating it.
- Balanced costs. A common strategy is to design the protocol such that each message has the same length, so that there are no incentives to send a different message than the expected one.

All BAR protocols are based on certain assumptions. Firstly, rational peers must be reluctant to risk. This means that they always follow the safest choice and that they are conservative about the impact of Byzantine nodes in the network. Hence, a rational peer never colludes with a byzantine node, just to obtain a greater benefit, if this puts in risk its participation in the system. Secondly, it is assumed that, if the protocol provides a Nash-equilibrium, all the rational nodes will follow it.

3.3.4 Monitoring Protocols

Monitoring data storage is concerned with the particular problem of enforcing correct storage on remote nodes. In a BAR model, both byzantine and rational peers may drop local stored data since it might be the best strategy. That requires monitoring the storing nodes for asserting if they continue to hold the information. If that is not the case, proper mechanisms can be activated to create new replicas and ensure data durability. A proof that a node is no longer storing the information can be used to apply penance measures.

The monitoring process consists on one verifier (V) that tries to prove that a storer (S) holds the data. For that, V sends some information that allows S to calculate a proof of possession. The following properties are considered when evaluating a protocol that implements this model:

Soundness - S cannot create a proof of possession without holding the original data. For this, the integrity of the data should be preserved.

Eligibility - V must be authorized to perform a verification.

Verifiability - Any other peer besides V can verify the correctness of the proof returned by S.

Efficiency - The amount of occupied space on V and the time of the operations should be minimized.

Monitoring protocols can be deterministic and probabilistic. In *Deterministic protocols*, nodes are required to provide a proof that they own the complete data[19, 14]. These protocols may require a large time for creating a proof and verifying it. In *Probabilistic protocols*, nodes are only required to prove that they hold parts of the stored data. Usually, data is partitioned in blocks. On each monitoring operation, the verifiers pick one or more blocks to audit. This choice is random to avoid storers from deleting the remaining blocks[36, 11, 25, 31]. The period between audit operations, the duration of the monitoring process, and the number of blocks audited determine the tradeoff between efficiency and soundness.

Several different monitoring protocols have been proposed in the literature. The approach described in [31] forces the transfer of the entire data or the entire block. This requires a significant amount of bandwidth and forces the verifier to also hold that information. Therefore, this solution may be inefficient. The approach described in [36] stores signatures of each block along with the original information. The verifier can request those signatures. This achieves verifiability, since any node can verify the signature, but requires extra storage space. Also, calculating digital signatures of large

data blocks is expensive. A variant of the previous scheme is described in [25]. It stores special blocks, called *sentinels*, in random offsets with respect to the beginning of the data. This approach relies on the secrecy of the sentinel's position. In each auditing operation, verifiers request that a subset of sentinels is sent by revealing their offset. Hence, the number of operations is limited and the probability of missing dropped or corrupted blocks grows with the amount of offsets revealed. A more sophisticated approach uses the properties of modular arithmetics, by using RSA-based calculations. Solutions that fall into this class[19, 14, 11] rely on *homomorphic* properties. More precisely, it is possible for the verifier to store a summary of the initial data while still providing a sound mechanism for proving remote data possession. The disadvantage of this approach is that it performs exponentiations and modules, which cannot be calculated efficiently.

Since these mechanisms are used to enforce data storage, their main use is on backup systems, where there is a direct reciprocity between nodes, i.e, each node must hold its partner's data. Hence, whenever one partner fails to reply to a challenge, it is possible to delete its data. This mechanism is very rigid, since nodes may crash and fail to reply to a single challenge due to reasons outside of its control. An alternative mechanism is based on grace periods[16, 5, 17, 10]. Here, nodes only allow the lack of replies until the grace period has expired.

3.4 Existing Solutions

This section gives a brief description of some of the most relevant solutions found in the literature, that consider data storage in P2P systems, reputation schemes, BAR protocols, and monitoring techniques.

3.4.1 Bittorrent

Bittorrent[8] is a partially centralized P2P file distribution protocol. To exchange content, peers establish tit-for-tat relationships. The entire network is managed by trackers, which are central servers that keep lists of downloaders for each file. The process of obtaining a file starts by contacting a tracker that returns a randomly selected set of nodes. The requester establishes a relationship with each member of that set. In each interaction, nodes proceed with an exchange of missing data pieces.

Interactions are managed by a local reputation mechanism based on download ratios. To maximize its reputation, peers are obliged to balance their download rate. With this, the protocol strives to achieve a pareto efficiency instead of a Nash-equilibrium. In economical analysis, this means that peers might commit themselves to obtain better download rates by establishing relationships only with nodes with higher upload rates. However, the protocol is not resilient to the problem of free-riding[39]. This happens due to the beginning of an interaction, when uploaders allow new downloaders to keep a low download rate. Rational peers can take advantage of this fact to continuously download content from different sources without reciprocating.

3.4.2 Oceanstore

Oceanstore[26] is a data storage system for ubiquitous computing, a new paradigm of distributed computing that strives to solve problems derived from mobile computing. This solution allows mobile devices to store data persistently on a P2P network of servers, while still being able of manipulating that information locally, even during periods of disconnection. Whenever the opportunity to connect with the servers arise, changes are committed and conflicts solved.

Data servers are organized in a structured P2P network. To perform search operations, two algorithms are used. The first one is a probabilistic approach, where nodes keeps an array of bloom filters for each neighbor in the network. A bloom filter is a special data structure that allows filtering search requests according to their identification. If the id of a searched data matches one bloom filter, the corresponding neighbor might know who holds the information. As an alternative mechanism, a variation of a distributed hash table is used.

Data survival is ensured through pure replication of writable data. The read-only information is stored using erasure-codes. To deal with simultaneous operations on the same data, replicas have to determine the final order of operations. To that end, a small group of servers starts a byzantine agreement protocol while the others apply a dissemination protocol to propagate the information. In the end, the final order is released to all the servers.

3.4.3 Samsara

In [32], the authors approach the problem of free-riding by relying on direct reciprocity relationships. Two particular problems are addressed. First, how to ensure that, in a P2P data storage system, nodes do not have incentives to delete the stored data. As in monitoring protocols, a challenge-response mechanism is used. This leads to the second problem of determining the appropriate action when nodes fail to reply to a challenge. A grace period is used to prevent unfair data deletion.

Regarding fair contribution, nodes are forced to locally garner storage claims of other peers. These claims consist on verifiable meta-data that have the same size of the information that is stored on each partner. This way, each peer is forced to locally store the same amount of information that it consumes. There is an inherent overhead to this strategy that is minimized by claiming-paths. These are transitive relationships between peers, where claims along the path can be replaced by normal data. For instance, if a node A stores a claim from B, that also holds a claim from C, then it is possible for B to replace C's claim by A's data.

Enforcing data storage requires bidirectional audits. Data owners use a simple challenge-response mechanism to monitor data storers, while, at the same time, storers audit data owners to ensure claims storage. When claiming-paths are created, these audits are forwarded along the path.

Samsara also deals with rational behavior when granting grace periods. Instead of waiting for the expiration of this period, nodes start deleting data blocks with an increasing probability. Special care is taken to minimize situations where crashed nodes unfairly lose their data, if they reply before the grace period expires.

3.4.4 BAR for Cooperative Services

In [3], the BAR model was firstly introduced and a general three-layer architecture is proposed for every solution that comprises the BAR model. Every step of a protocol is an exact Nash-equilibrium such that it is of the node's best interest to follow the expected behavior. A prototype for data backup is described as a proof of concept for the BAR model. In this system, nodes periodically challenge their partners to ensure data durability. The utility function considers the benefit from backing-up data, and includes communication and storage costs. It is assumed that nodes do not collude.

Peers are organized in a structured overlay with static and full membership. Each operation is seen as a command that is proposed to all the participants and an agreement is used to decide the output of each action. Instead of relying on an implementation of the byzantine agreement protocol, each message is broadcasted using a Terminating Broadcast primitive (TRB). This is an algorithm that ensures that the message is either delivered to every correct nodes or dropped, otherwise. This choice was made in order to reduce the set of possible outputs that a rational node can create when proposing a value in the byzantine agreement.

To tolerate byzantine behavior, each message is signed and follows a strict format. Therefore, any incorrectly formatted message presents a proof of misbehavior (POM) and is sent to every node using the TRB primitive. When it is not possible to prove that a note misbehaved, peers can insert suspected nodes into a bad-list. This list is exchanged periodically and when a node is included on a quorum (majority) of bad-lists, it is classified as byzantine and removed from the system.

Enforcing fair participation requires that all participants have the opportunity to submit a command. This is done by dividing the time in operations and rounds. Each participant submits orders when elected as the leader of the first round of an operation. This election is achieved by using a verifiable pseudo-random number generator, such that no node can manipulate the election, and the process is verifiable. The first leader of an operation has the ability to submit a totally new order. If it fails, another leader is elected for the next round. To prevent nodes from ignoring steps of the protocol, a message queue for each user is locally stored on each peer. Whenever a message is expected from a user, a hole is introduced in its queue, so that it cannot send another message until it dispatches the expected one. All the messages are balanced such that the expected behavior is never more costly than any alternative, incurring in extra bandwidth costs. To deal with message delay, penance messages must be sent periodically, removing any benefit obtained from that delaying.

Another mechanism is implemented to force nodes to perform the expected work, as answering to requests or periodically sending messages. More precisely, a trusted witness mediates every interaction, and ensures that each node operates as expected. Since every message is broadcasted, the full membership provides an implementation for that witness. This mechanism combined with message queues denies service to a node that fails to send any message.

In the BAR backup prototype (BAR-B), all the nodes possess a certain quota of storage on other nodes. This is ensured by periodically exchanging storage information about the neighbors, and hence disseminating information about possible unbalanced relationships. Concerning the challenge-response mechanism, a storer must return a

certificate that proves data storage. Each block of data possess a validity controlled by a lease. Later, when the storer is challenged, it must provide a valid reply, a proof that data was deleted or that the lease has expired. Any other reply constitutes a POM. Non-responses are avoided by the global witness mechanism.

3.4.5 BAR Gossip

BAR Gossip[29] uses the BAR model to tolerate both byzantine and rational peers in a P2P gossip protocol. In this type of protocols, peers exchange information with a random set of neighbors in each interaction. This ensures, with high probability, that, after a sufficient number of interactions, data is spread across the network. BAR Gossip creates an exact Nash-equilibrium for nodes to follow every steps of the protocols and to ensure fair exchange of data. It is assumed that nodes do not collude.

As in [3], each node keeps a full and static membership. A trusted central server called tracker is responsible for starting the dissemination process. Data is partitioned in several blocks. Initially, the source selects a random group of peers and sends different subsets of blocks of data to each selected peer. Then, for a certain number of rounds, each peer choose other neighbors to exchange missing information in a fair manner.

Byzantine peers have several opportunities to attack the system by conditioning the dissemination process. To deal with this problem, in each round a pseudo-random number generator is used to select each node's partners. This also has the aim of mitigating collusion and denial of service. In addition, each message is signed so that, as in [3], any incorrectly formatted reply constitutes a POM. These proofs are spread by sending it to the source, who will disclose an eviction order. To provide incentive for sending POMs to the origin, the sender is included in the next set of users that receive all the updates in the initial round.

The system strives to avoid free-riding by demanding a balanced exchange between peers. Partners calculate a set of updates that each one is missing, send them ciphered with session keys, and only then they exchange session keys. This delay is to give incentive for rational nodes to obey all the steps of the protocol, in an approximate solution to the Fair Exchange problem. The fact that nodes still avoid sending the encryption keys is mitigated by a penance mechanism. A node that fails to send the key is bombarded with messages. These extra costs may dissuade nodes to misbehave.

The system also addresses the possibility of nodes lying about their history, to avoid spreading updates. With the balanced approach, peers obtain the same amount they send, so they are encouraged to exchange as much as possible. In the unbalanced protocol, a node may under-report the updates it has available. BAR-Gossip does not solve this problem, but relies on the fact that under-reporting increases the risk of receiving a repeated update. Whenever a node promises to send an update and does not meets that promise or when it sends invalid data, this will be used as a POM against it, since all messages are signed. Any invalid session key that is traded is also used as an accusation alongside with the ciphered information.

3.4.6 Flightpath

The authors of BAR Gossip[29] proposed several improvements on the previous solution to allow dynamic membership, while still keeping a full view. Unlike the two other BAR implementations[3, 29], an approximate Nash-equilibrium is used to obtain better performance.

Besides being the source of information, in Flightpath the tracker is also responsible for managing membership by constantly pinging all the participants. Special care is taken to deal with churn. More precisely, the authors try to avoid the situation where a significant amount of peers join the network simultaneously and, since they do not have nothing to exchange, they overwhelm previously joined nodes with requests for new updates without contributing for the system. Two solutions are proposed. One is to divide time in epochs. Each epoch is associated with a list of members. When a node joins the network, the tracker places it in the epoch $e + 2$ where e is the current epoch. When moving from e to $e + 1$, the tracker disseminates the list of members from the new epoch. This solution is concerned with how to spread membership information in an efficient manner. However, it does not allow a recently joined peer to immediately start receiving updates. That problem is solved with the Tub algorithm. Here, the list of members is ordered by node's entrance in the system and partitioned into groups called tubs. Recently joined peers are placed in the most recent tub. Then, they create a view of its potential partners with whom they can exchange updates. That view is composed by all the members of the node's tub and by an exponentially decreasing number of peers from the older tubs.

This protocol uses the same strategies of Bar Gossip for preventing free-riding. However, it allows nodes to establish concurrent trades with multiple partners. This introduces the problem of a node being overloaded with concurrent requests. To mitigate this, a mechanism of reservations is used to limit the total number of neighbors with whom each node communicates. Additionally, the number of updates traded is limited so that different concurrent neighbors don't send a repeated update. Instead of trading all the missing updates, each neighbor sends a subset of it. Still, there is the possibility of some missing updates not being exchanged. Erasure-codes[37] are proposed to mitigate this problem. Moreover, slightly unbalanced exchanges are allowed. Still, each node keeps an imbalance ratio for each of its neighbors, that is lower bounded, to avoid free-riding.

3.4.7 Cooperative File-Backup System

CBS[16] is a P2P cooperative file storage system where nodes provide storage space to its neighbors and replicate their data in other peers. It relies on direct-reciprocity to address free-riding and implements a simple retrieval of information mechanism for ensuring remote data storage.

This solution is partially centralized since a single server is used to perform node search. Whenever a node wants to backup its data, it contacts the central server for potential partners. Erasure-codes are used to introduce redundancy. The redundancy factor r determines the number of relationships each node needs to establish, to ensure data durability. Here, it is possible to add or remove partners from its relationships. It

also allows a dynamic membership, where nodes can enter and leave freely.

To tolerate free-riding, it relies on direct reciprocity as a way to enforce fair-exchange. When establishing new relationships, nodes define the amount of space they have to exchange and the fraction of time that each one has to be available everyday.

To ensure that partners holds the data for a sufficient amount of time, a probabilistic approach is used. In each audit operation, verifiers request for the transfer of an entire randomly chosen data block. When dealing with consecutively lack of responses, peers wait for the expiration of a grace period of two weeks until they drop the data that belongs to the evader, to prevent that an honest but failed node loses its information.

This protocol considers a possible attack to the grace period which happens when peers deliberately drop its partner's data and fail to respond until the grace period expires, picking a new set of neighbors. Three different strategies are used. The first one is to force a node to waste space that eliminates any benefit that comes from deleting the initial data. Other approach consists on a prepayment for remote usage of space, that also cancels any possible benefit from not storing its partner's data. A post-payment scheme is also used with the same intent. Here, a central server monitors each interaction and, whenever a node misbehave, its neighbor can issue a complaint that forces both peers to pay an extra storage time for the arisen of this conflict.

3.4.8 Remote Integrity Checking

In [14], two deterministic approaches to the problem of enforcing remote storage and integrity are proposed.

The first strategy is a simple challenge-response scheme. The verifier creates a random challenge that is sent to the storer. Then, it returns the digest of the concatenation of the data with the challenge. It is necessary for the verifier to hold the entire data so that it may verify the response.

The second approach is based on the homomorphic property of RSA-based calculations. The file is seen as a huge number m. The verifier generates $N = p * q$ where p and q are two large primes that must be kept secret. Based on Euler's theorem, it is true that $a^M \ mod \ N = a^m \ mod \ N$, where $M = a^m \ mod \ L$, $L = (p-1) * (q-1)$, a is a random number and mod is the module operation. Only p, q and L are kept secret. During an audit operation, the verifier sends a challenge r to the storer. It then generates $A = a^r \ mod \ N$ and $B = A^m \ mod \ N$. The verifier generates $C = M^r \ mod \ N$ and checks if $C = B$, which must be true under Euler's theorem. The important aspect about this approach is that the verifier only has to store M, which is usually significantly smaller than m.

3.4.9 Probabilistic Cooperation Assessment

The authors of [36] propose a probabilistic approach to ensure data storage in an ad-hoc network. The idea is to allow nodes to establish connections with nearby peers in order to backup data. A reputation mechanism is applied to motivate nodes to contribute to the system.

Peers create ad-hoc connections with nearby computing devices. After that, they can pick any node to backup its data and either audit directly or delegate that responsi-

bility to other peers. To maintain the reputation of nodes, the mechanism for auditing data is verifiable, which means that any peer can check whether the storer is fulfilling its obligation or not. This allows an indirect interaction between the verifier and the storer such that responses can travel through several nodes, adjusting the reputation of storers on those nodes.

Verifiers generate digital signatures for each block. When monitoring the activity of storers, it is required the dispatch of the signature that is associated to a randomly chosen block. As in CBS[16], a grace period is used to deal with lack of responses. However, it addresses a problem not dealt by other monitoring protocols, which is the periodicity of audit operations. This is done through a payment with reputation points for each operation.

3.4.10 Provable Data Possession

PDP[11] is a monitoring protocol that relies on the homomorphic property to perform probabilistic proofs of remote data possession. As in the RSA-based approach of CBS[16], the verifier only stores a small amount of information that allows it to verify the responses from the storer. The storer keeps homomorphic tags for each block, which is a small piece of meta-data. These tags combined with a challenge and the original data block prove that the information is being held correctly. The protocol is probabilistic because it only verifies a subset of data blocks. A pseudo-random number generator is used to select which are the blocks that must be verified on each operation. To deal with the inefficiency of exponential operations inherent to RSA-based homomorphic functions, an optimization is proposed. Instead of exponentiating each block with the challenge, it is allowed to avoid that operation. This is a tradeoff between efficiency and soundness because, with this optimization, the storer can only hold a digest of the information and still provide valid proofs.

3.4.11 Fireflies

In [24], the authors propose a new protocol for creating a P2P structured overlay that is tolerant to byzantine intrusions. This is done by structuring the overlay as circular relationships called rings, where a node is connected to only two other peers. Nodes are positioned in each ring according to the hash of the concatenation of the node's id with the ring number. This way, peers cannot chose with whom they are neighbors, mitigating collusion, sybil, and eclipse attacks. Constantly monitoring the liveness of each node, this protocol allows dynamic and full membership without using a central server. The monitoring process consists on periodic ping operations on every live successor in each ring. Whenever there is a timeout, the monitoring node emits an accusation. If the defendant peer is alive, it will receive that accusation and generate a counter-proof that will keep him seen as correct in the network.

To tolerate f byzantine faults, the overlay must contain $2f + 1$ rings. Only in this way, it is ensured with high probability that a node is directly connected to a majority of correct nodes. The fact that any node can send an accusation allows byzantine peers to attack their successors. The counter-proof prevents a node from being unfairly convicted but it does not avoid denial of service attacks by continuously sending ac-

cusations. Fireflies mitigates this problem by including on each message an activation buffer that indicates for each ring if the predecessor of the source of the message can still generate accusations. However, since it is assumed that the communication channels are not reliable, it is possible that some of the predecessors are correct and still emit accusations, due to message loss in the communication channel. In this case, the corresponding rings will be deactivated. Hence, some of the predecessors belonging to active rings may be byzantine. To prevent peers from being overwhelmed with byzantine predecessors, nodes are only allowed to deactivate f rings. This ensures with high probability that, in any moment, at least one of the predecessors is correct. Though, this approach does not completely prevent denial of service attacks since it is not necessarily true that all the rings that contain byzantine predecessors are deactivated. Thus, some byzantine predecessors may still emit false accusations.

4 Proposed Architecture

In this section, we propose an architecture for implementing BARRAGE, a middleware for storing data in a P2P network. The basic model consists of execution tasks, such that each task must be executed by a different node. Some tasks produce data that is consumed by other tasks. Hence, there must be nodes that produce data, called *producers* and nodes that consume data, called *consumers*. In BARRAGE, we aim at allowing the producers to leave the network, before transferring the data to the consumers. This can be achieved by storing data on another type of nodes, named *storers*. In one storage operation, nodes can only adopt one role (producer,consumer or storer).

The main goal of this thesis is to develop a robust sub-system that allows the producers to store data on storers, and the consumers to retrieve it, preserving the integrity of the information, despite the existence of faulty nodes. It is assumed that any other mechanism external to the sub-system is managed by a trusted third party. By external mechanism we mean distributing the identity of all the participants to all the nodes that require it, as well as any cryptographic information, like public keys. Furthermore, scheduling of storage operations and data deletion are also outside the scope of this work. In addition, communication channels are considered reliable and we assume that the system is synchronous, so that the time to conclude any operation may be upper-bounded.

To model the behavior of nodes, we use the BAR model. Therefore, we classify peers as byzantine, rational, or altruist. In the context of our basic model, producers may fail to produce the data, generate wrong information or simply fail to dispatch it to any storer. Storers may crash and loose all the data, they may deliberately delete it or repudiate its reception. Consumers may also crash or repudiate data made available by the storers. In addition, nodes may collude to defeat the system.

Since we are also considering rational peers, we need to identify the utility function that models their behavior. BARRAGE is intended to be integrated with a P2P grid system. Therefore, our model is inspired by BOINC[4], that demonstrates that rational nodes volunteer if they receive an incentive. More precisely, nodes are classified in a ranking system, according to a number of points, obtained for contributing to the system. Hence, producers receive points for generating correct data and store it on

another nodes. Storers are rewarded for correctly holding the data. Points are also given to the consumers when they report to have consumed correct data. In addition, negative points are given as punishment for misbehavior. We delegate the responsibility for managing this process to the trusted third party, based on information provided by our protocol. Other type of mechanisms are applied to persuade rational peers to contribute to the system. More precisely, balanced costs are used to ensure that a rational peer always benefit more from following the protocol than from deviating from it. Non-determinism is minimized to limit the number of available alternatives.

To tolerate byzantine faults, it is assumed that there are at most f byzantine participants among all the storers, producers, and consumers. Forcibly, there must be several producers for the same information, as all the f byzantine peers may belong to that group. We argue that the lower bound for the number of producers is $2f + 1$ so that there is, at least, a majority of identical signed versions of the result, in order to reach a conclusion about its correctness. In normal circumstances, producers would not be byzantine, and storers would only fail by crash or stop. Then, only one producer would be required to produce correct results, and it would be necessary for it to replicate the result to different storers, in order to tolerate any fault by crash on storers. In our model, since there are several producers, then replication is already achieved if each producer stores its results in a different storer.

At the time the consumers retrieve the data, a majority must equally hold where gathering the data from the storers. With one to one relations between producers and storers, the worst case happens for the maximum number of relations with at least one byzantine participant, i.e., when each byzantine node is related to a correct peer. Since the sum of byzantine producers, storers and consumers is lower or equal to f, by assumption, there will be, at most, f incorrect relations. Considering a total of $2f + 1$ interactions, at least $f + 1$ (a majority of) copies of data will be correctly produced and stored on non-byzantine storers. Therefore, the number of peers belonging to each group must be no lower than $2f + 1$ (for a total of $4f + 2$ nodes in a producers-storers interaction).

In addition, it is required to distinguish between correct and incorrect behavior. This can be done by defining a strict format for each message and enforcing the usage of digital signatures. Therefore, any incorrectly signed information can be used as a Proof of Misbehavior (POM). We leave to the central entity the responsibility of taking appropriate measures when a POM is generated. It is important to notice that faults by omission do not allow to create POMs, since there is no way to distinguish between crashed nodes and nodes that deliberately avoid sending information.

The rest of the architecture is described in two phases. Firstly, we assume that the duration of the data hold by storers is upper-bounded. Then, that assumption is removed. In the end, we propose a form of electing peers for the different roles with the aim of avoiding collusion.

4.1 Short-duration Storage

In this first scenario, it is assumed that there is an upper bound for the time between the moment when producers store the data and consumers request it, and this bound is sufficiently short so that correct peers do not leave the network. We also assume that

there will be, at most, f failures and that rational storers know, a-priori, that data will be retrieved in a relatively short time. Points will only be given if data is hold until the consumers request it. So, the protocol should guarantee that the only rational behavior that strives to maximize the number of points is to avoid discarding the data, until the central authority authorizes it.

A major issue arises when the consumer tries to retrieve the data, but it is not available in a certain storer. Two possible events may lead to this situation: i) a producer failed to store the data, or ii) the storer dropped it. The consumer should be able to determine who misbehaved in order to apply punishment measures. A possible solution consists in ensuring that the information was stored, proving that the storer deleted it, or that the storer did not receive it, proving that the producer misbehaved. This can be done by requiring the producer to send the data and the storer to return a proof of receipt. This is a classical Fair Exchange problem, only solvable with a trusted third party. We chose to appeal to it only in situations of misbehavior called conflicts, for performance reasons. We argue that solving these conflicts demands expensive operations that discourage rational nodes to misbehave, with the intent of avoiding these costs. However, the expenses of accusing another node should never be greater than the benefit obtained from successfully delivering the data to the consumers. This is relevant, to ensure that rational peers still have incentives to accuse byzantine nodes when they cause conflicts.

4.2 Long-duration Storage

In the previous scenario, it was assumed that there was an upper bound to the time between data storage and retrieval. In some situations, that assumption may not hold. Firstly, the information may only be required much after it was produced. Secondly, it may be necessary to hold the information for an undetermined period of time, as a backup mechanism. With these new requirements in mind, we now remove that assumption and embrace the need for storing data for an arbitrary large period of time, while still assuming that, at most, f nodes are byzantine.

In P2P networks, the greater delay between data production and consumption might prevent storers from remaining in the system for the entire period, to be rewarded. The outcome is that no rational storer will ever be interested in participating in the system. One possible approach to solve this problem requires giving points to peers for each period of time that they hold the data, even if they do not remain alive until its final consumption. A simple query mechanism by questioning the storers about the data would not suffice. Instead, it is necessary to periodically perform a sound verification on the integrity of data, i.e., it must be proven that storers hold the initial data, while still ensuring data recovery.

This role must be given to a special type of nodes: the *monitors*. They send a challenge to the storers, that reply with a signed correct answer. For each correct auditing performed over a storer, they are rewarded with points. This strategy may give incentives for these monitors to collude with storers, by falsifying replies. Therefore, our verification mechanism should be sound, by proving complete data possession and withdrawal; and verifiable, so that the ranking system might verify if it is performed correctly. Additionally, it should require the least possible available storage space from

the monitors, to not dissuade them from performing this role.

There is an interesting approach, based on homomorphic hash functions[11, 14], that allows the creation of challenges from a summary of the original information and a secret value. This minimizes the amount of storage space wasted on each monitor. However, if the storers obtain the secret value, they are also capable of holding only the summary of the data while still providing correct responses to the challenges. The problem is that this summary is irreversible, which means that the original information cannot be recovered. Since storers have incentives to minimize the amount of storage space, and monitors want to avoid the costs of creating challenges and verifying responses, then the rational strategy will be the collusion between nodes. This can be solved in two ways. The first is by avoiding long-term relations between data monitors and storers, so that monitors do not have any other motivation to collude with the storers besides minimizing costs. The second is by using private keys as secret values that must be critical to the participation of the nodes in the system. Only then, the monitors will not have incentives to give the keys to the storers. These keys must be shared between monitors and the trusted third-party, so that the mechanism can be verifiable.

An important remark must be made about the verification of these audit operations. If each challenge-response forces the central component to validate the actions performed by the monitor, it would be better not to have any monitor at all. Instead, points should only be given after a fixed period of time. Each monitor has to send the summary of the information and all the challenges and respective responses received during a certain period. To minimize the overhead of verifying all the answers, the central component might only request a subset of them. It must be ensured that the monitor cannot only generate the requested challenges. A possible approach is to generate each challenge from a monotonically increasing order number. Then, the ranking system only requests the challenges associated to a random subset of order numbers, preventing the monitor from guessing which challenges it should produce. From a rational point of view, the best strategy might be to generate all the expected challenges, to avoid unnecessary risks.

Finally, there is another issue related to rational storers. To minimize the storage overhead, they might collude in order to partition the correct data into smaller pieces. Each piece is stored in a different storer. Whenever an audit is performed, they regather the original data and return a valid response. The communication costs might dissuade nodes from adopting this behavior. However, in long term storage, it might be profitable to apply this strategy than to permanently hold the entire information. Such scenario might make the system susceptible to byzantine failures. One possible solution consists in forcing each storer to hold the data formated in a unique way. For instance, by storing the data ciphered with a unique key. Ciphering costs can be made sufficiently high to dissuade storers from performing them on each audit operation.

4.3 Role Selection

All the previously proposed mechanisms are based on the assumption that the majority of nodes do not collude. This can only be achieved if the process of role selection is outside the control of any peer.

To this end, we plan to organize nodes in a structured overlay. More precisely, we

intend to adapt the byzantine fault tolerant overlay of Fireflies[24] to BARRAGE. As it was described in Section 3.4, this overlay organizes nodes in several rings, which creation is outside the control of the nodes. So, a possible form of electing monitors for each node is to select its predecessors in the rings. Producers, consumers and storers may be selected by choosing the peers that are closer to pseudo-random numbers, generated by the trusted entity.

5 Evaluation

The main goal of this work is to implement a P2P storage sub-system that is BAR-Tolerant. In other words, the system must guarantee safety and liveness in the presence of a number of byzantine nodes that is bounded, and up to N rational peers.

To assess the performance and correctness of our protocol, we plan to perform an analytical analysis of our solution. More precisely, we intend to take into consideration the following aspects:

- Analyze the effects of certain types of byzantine failures, with special emphasis given to *denial of service*, *collusion*, *data deletion*, and *transient* and *permanent* failures;
- Calculate the consumed *bandwidth*, that depends on the number of messages and their respective sizes;
- Determine *storage overhead*, introduced by the monitoring mechanism;
- Calculate the *complexity* of *data storage*, *data recovery*, and *conflict resolution* operations;
- Determine the *probability of data loss* according to different levels of the availability of nodes and different fractions of byzantine and rational peers;
- Use a methodology to determine if the protocol is a Nash-Equilibrium, based on Game Theory concepts.

If time permits, we will also perform an experimental analysis, in order to improve our work by measuring metrics that cannot be easily evaluated by the analytical approach. Namely, we expect to determine the relative costs of the *amount of computation* related to cryptographic calculations and conflict resolution with regard to storage and communication costs. We also aim at evaluating the impact of making the protocol BAR-Tolerant. For that end, we plan to compare two different versions of the protocol: One with mechanisms that enforce the principles of the BAR model, and the other without them. As a basis for comparison, an application that demands the computation of a significant amount of storage space should be run on both versions of the protocol. For instance, we may run an implementation of the Monte-Carlo methods for simulating the behavior of magnetic molecules[3].

[3] http://qah.uni-muenster.de/

6 Schedule

Future work is scheduled as follows:

- January 7 - February 29: Detailed design of the proposed sub-system.
- February 29 - March 29: First implementation of BARRAGE
- March 29 - April 15: Perform the complete theoretical analysis
- April 16 - May 14: Finish implementing the prototype and perform the experimental evaluation
- May 15 - May 31, 2011: Write a paper describing the work and contribution of the dissertation.
- May 24 - June 13: Finish the writing of the dissertation.

7 Conclusions

This document has, in the first place, presented a summary of the state-of-art related to P2P storage and the BAR model.

We started by presenting an overview of P2P architectures, including its main problems and design issues related to building storage systems. Then, we introduced the different models for dealing with different types of nodes behavior, and protocols that comprise them. Among these models, we addressed byzantine fault tolerance (BFT), crash by failure or stop, incentive-based models, and BAR tolerance (BAR-T). The related work ended with a description of several relevant solutions from the literature.

We also have proposed an architecture for providing data storage generated by producers, in a P2P grid system, and consumed by other nodes. To model the behavior of peers, the BAR model was considered. We identified some of the major issues related to the main interactions between peers, and proposed mechanisms to solve them.

We concluded the report with a description of the methodologies that we expect to apply during the evaluation of the proposed architecture.

Acknowledgments

We are grateful to João Leitão, Oksana Denysyuk, and João Fernandes for the fruitful discussions and comments during the preparation of this report. This work was partially supported by the project "HPCI", under the FCT grant (PTDC/EIA-EIA/102212/2008).

References

[1] E. Adar and B. A. Huberman. Free riding on gnutella, 2000.

[2] A. Adya, W. J. Bollosky, M. Castro, G. Germark, R. Chaiken, J. R. Doucer, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment.

[3] S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *SOSP'05 20th ACM Symposium on Operating Systems Principles*, pages 45–58. ACM, 2005.

[4] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[5] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pstore: A peer-to-peer backup system. 2006.

[6] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66. Springer-Verlag New York, Inc., 2001.

[8] B. Cohen. Incentives build robustness in bittorrent, 2003.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. OSDI*, pages 177–190, 2006.

[10] O. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI: Symposium on Operating Systems Design and Implementation*, pages 285–298, 2002.

[11] A. B. R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores.

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

[13] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[14] Y. Deswarte, J.-J. Quisquater, and A. Saïdane. Remote integrity checking.

[15] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, 2001.

[16] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. Cooperative backup system.

[17] M. L. Fakult, F. F. Informatik, and T. U. München. Peerstore: Better performance by relaxing in peer-to-peer backup, 2004.

[18] B. Garbinato and I. Rickebusch. A topological condition for solving fair exchange in byzantine environments. In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS'06), volume LNCS*. Springer, 2006.

[19] E. L. Gazzoni, D. Luiz, G. Filho, P. Sérgio, L. M. Barreto, and E. Politécnica. Demonstrating data possession and uncheatable data transfer. Technical report, 2006.

[20] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *PROC. ACM SIGCOMM*, pages 147–158, 2006.

[21] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03*, 2003.

[22] R. Gupta and A. K. Somani. Reputation management framework and its use as currency in large-scale peer-to-peer networks. In *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, P2P '04, pages 124–132, Washington, DC, USA, 2004. IEEE Computer Society.

[23] G. Hardin. The tragedy of the commons. *Science*, xx:1243–47, 1968.

[24] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. *SIGOPS Oper. Syst. Rev.*, 40:3–13, April 2006.

[25] A. Juels and B. S. Kaliski. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. ACM, 2007.

[26] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. pages 190–201, 2000.

[27] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[28] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. Flightpath: Obedience vs choice in cooperative services. In *OSDI 2008*, 2008.

[29] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 191–204, Berkeley, CA, USA, 2006. USENIX Association.

[30] J. Liang, R. Kumar, and K. W. Ross. The kazaa overlay: A measurement study. *Computer Networks Journal (Elsevier*, 2005.

[31] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 29–41, 2003.

[32] L. C. Lpcox, L. P. Cox, and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 120–132. ACM Press, 2003.

[33] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC*, pages 311–325. Springer-Verlag, 2002.

[34] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. pages 31–44, 2002.

[35] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, September 1951.

[36] N. Oualha and Y. Roudier. Securing ad hoc storage through probabilistic cooperation assessment. *Electron. Notes Theor. Comput. Sci.*, 192:17–29, May 2008.

[37] J. S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software – Practice & Experience*, 27:995–1012, 1997.

[38] C. D. S. Services. The gnutella protocol specification v0.4.

[39] J. Shneidman, D. C. Parkes, and L. Massoulie. Faithfulness in internet algorithms, 2004.

[40] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[41] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer file sharing. *3rd Symposium on Networked Systems Design & Implementation*, 2006.

[42] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, UC Berkeley, April 2001.