Reliable and persistent storage for CoDeS a distributed collaborative system.

Youssef R'kaina

Advisors : Dr. Joan Manuel Marquès Puig
Dr. Daniel Lázaro Iglesias

Universitat Oberta de Catalunya

Abstract

One of the most important recent developments in computing is the growth in distributed systems, these systems are designed to ensure features as self-organization, availability, scalability, fault-tolerance, consistency among others. With the huge success of Internet and its continuous development and the fast progress in the domains of processors, hard-drives and networks, new paradigms of information and computing systems emerged in response to new ways for users to interact, communicate, collaborate and use data. A whole new vision have come to light to offer new systems more optimized, robust and aware of the needs and requirements of the future.

Data Grids, cloud computing and peer-to-peer networks are some of the distributed systems that appears as a result of the enormous research done in this domain. They are aimed to specific fields and requirements, however they share a lot of common aspects and interests. Data Grids are more used to exploit resources to execute computational tasks, Cloud computing provide large data centers and services to end users in return of an amount of fees and finally peer-to-peer networks are destined to applications sharing disk space and bandwidth.

To address the lacks of cloud computing, desktop grids and peer-to-peer networks to use non-dedicated resources for general purpose computing, contributory computing system emerged. This new system is responsible of aggregating resources that will be used as a platform where services can be deployed allowing a collective use of them. Offering services is the main goal of a community implementing contributory computing. CoDeS [22] is a distributed collaborative systems that has been designed and implemented to prove the technical feasibility of this new concept. It offers a mechanism for service deployment, a resource discovery mechanism, and an availability-aware resource selection mechanism. All these mechanisms allow the creation of contributory communities. However, it doesn't have a mechanism for distributed storage. This research is aimed to find a solution for a distributed storage system adapted for CoDeS. By studying how DSSs work and how they are implemented, we can conclude how we can implement a DSS compatible with CoDeS requirements.

Contents

1. Introduction

1.1. Background and Motivation

Storage has always played an important role in computing. At the beginning, storage systems were destined to individual machines, however with the rapid evolution of storage hardware and network technologies, new challenges and new needs arouse. The main goal of DSSs is to aggregate several resources in a simple domain or all over the Internet to exploit them in order to overcome their underutilization and to offer means for collaboration between the different users of the distributed system.

These systems need to ensure some features to be able to respond to the requirements of this new paradigm. One of these features is self-organization. To minimize the manual administration required for such systems, the distributed system must be designed to be capable of automatically handling administration tasks and adapting the system according to utilization patterns. Another feature is availability. Even if the data is shared among several machines, the user must see it as if it were in its own machine, thus it must offer simple interfaces for end user applications to access data and services in the same way as in the case of a single machine. Scalability - when we talk about distributed systems, we talk about thousands of interconnected machines all over the world, these machines can join and leave the system whenever they want. The system must not only ensure a good degree of performance in the case of a limited number of machines but must maintain this degree of performance in the case of the joining of an increased number of machines. Fault tolerance - the system must be capable of handling machines disconnection and systems failures by keeping a stable state of the system and without the knowledge of end users, in other words these failures need to be transparent for the user. Consistency – DSS must ensure consistency even in the presence of events which challenge it such as concurrent updates. There are two types of consistency: strong and optimistic. The primary aim of strong consistency is to ensure data is viewed and always remains in a consistent state. In the case of optimistic consistency, the primary purpose is to keep data consistent without imposing the restrictions associated with strong consistency. These are only examples of features offered by DSSs and this document will introduce others.

DSSs have evolved from providing a means to store data remotely, to offering innovative services like publishing, federation, anonymity and archival. This document will present a taxonomy for DSSs and for each element of the taxonomy, we'll provide its description and the existing solutions that implemented it. We'll present also the solution that we consider suitable to be integrated with CoDeS and the proofs that we've elaborated in order to support our suggestion.

1.2. DSS for Contributory Computing

1.2.1. General Definition

CoDeS [22] is a contributory computing system which is a new model of distributed systems, it's responsible of aggregating resources that will be used as a platform where services can be deployed. These later will be used collectively by the community. Offering services is the main goal of a community implementing contributory computing. The idea of contributory computing has been developed to overcame the limits of others distributed systems as cloud computing, grid computing, desktop grids, volunteer computing or peer-to-peer systems.

To contribute resources to the community, several policies have been defined to describe how much resource could be used and how much time they could be available. For example, users could allow a fixed amount of resources that will be available as soon as their host is connected to the network. Members of the community are free to decide which policy they want to apply and they can stop their contribution whenever they want. Another point that need to be taken into account in a contributory system is the fact that resources are not strictly dedicated and not always available. Giving that contribution depends on members will, they all have to share common motivations to achieve the community objectives. The amount of resources allowed to each member is not restricted, because the resources of the community are considered to be used collectively and not individually.

A contributory community relays only on the resources shared by its number, thus there are not dedicated and centralized hosts. It must work in a decentralized mode knowing that each nodes could be disconnected at any time. The members need to bear in mind that for the contributory system to work they imperatively must share enough resources for the services to work.

There are several scenarios where a contributory community could be built. Classification of these kinds of systems are based upon two factors: the membership policy and the services that could be deployed. The first factor means that we can decide that membership will only be authorized for a specific group of people, the second factor on the other hand refers to the type of service that could be deployed, in other words, we can allow the deployment of every type of services or we can restrain the deployment to a specific kind of services beforehand upon entrance in the community.

The implementation of the contributory system is a middlewear that must be installed in each node and will handle autonomously the aggregation and the management of the contributed resources. Users can control the amount of resources that they want to share and when to do it, thanks to this middlewear. Additionally, this later offers a service deployment platform.

1.2.2. Example Scenarios

The main scenario that the storage system has to enable is service deployment in a contributory community managed by CoDeS.

There are two main cases where files are involved:

- Service Specification:
  - The specification that is required by CoDeS in order to deploy a service.
    - Details specified in [22].
  - Needed by ServiceManagers [22] to deploy services.
    - There are a number of SMs per service.
    - Each one needs to know the Service Specification and act in consequence.
      - All need to know the same.
  - Can change over time (e.g. change a service configuration parameter like number of replicas, activate or deactivate the service).
  - Small.
  - Read/modify whole file at once.
- Executable files
  - Needed by worker nodes to execute a service [22].
  - Immutable (new versions should have different names).
  - Variable size (could be very large).
  - Read whole file at once.

1.2.3. Features

Here are the features of a DSS for a contributory system:

- Exploitation of unused resources: all resources of a single machine are not used totally all the time and there is a significant amount of resources unused for long periods. A contributory system will resolve this problem by manipulating these resources when they are not used.
- No need for investment in dedicated resources: giving that all the resources needed are contributed by the community.
- Amount of resources adaptable to load: the amount of resources available in the community will be proportional to the amount of resources contributed by its members, thus the system need to ensure an equilibrium to prevent an overload of the community.
- Collective use of resource: the main goal of the community is to reach its objectives by offering a platform for service deployment, thus members need to be aware that these services are not meant for individual use but for a collective use.

1.2.4. Requirements and Challenges

In the current version of CoDeS there is only a centralized storage system, used to help proving the feasibility of the other modules responsible for the distributed deployment services system. The main objective of this research is to provide a distributed storage system that will replace the existing one and will help in the design and development of a complete distributed contributory system. To provide a distributed storage system, we need to bear in mind a set of functional and non functional requirements that define a contributory computing system and which represent the boundaries of the system. In the following paragraphs, we'll present the set of the two type of requirements.

Here are the non functional requirements and challenges of a DSS for a contributory system:

- Self-organization and self-management: management and configuration of the DSS will be restrained only to each node. The function of each node must be decided autonomously by the contributory software. Nodes must be able to join and leave the community in a transparent way.

- Decentralization: it's forbidden to decide that a node will be used as a central point of the system. Each node could support any task and it can be selected dynamically and autonomously by the system.
- Failure tolerance: The DSS must deal with the fact that any node has the right to leave the community at any time even if it's responsible for a service.
- Scalability: The DSS must be able to scale up to a lot number of nodes and guarantee a good quality of service.
- Data availability: The DSS must ensure the availability of its data. The response time to get data must be reasonable in accordance with the resources available in the community. Replication, reactive repair and prediction are mechanisms used by the system to ensure availability.
- Ability to deal with Internet-distributed resources: the system must be able to deal with long latencies and reduce bandwidths.
- Ability to deal with heterogeneous resources: The resources contributed may differ from their capacity to their behavior. The system must be able to select the fitting resources for each task among all those that form the community.
- Security and privacy: Monitoring maliciously interactions and user activity or access illegitimate data are some of the security failures that the system need to deal with. It must incorporate mechanisms that guarantee the security and privacy of its users.

Here are the functional requirements :

- Save files of variable size
  - Large files: immutable
  - Small files: mutable
  - “Atomic” modification:
    - The reader is certain that the retrieved version is the last one can be done e.g. by modifying all replicas at once or by reading with quorum.
  - Read whole file at once.

Another aspect that needs to be respected when designing the DSS is the high level design of CoDeS which focus on properties as modularity and extendability. The prototype of CoDeS was implemented in order to prove the feasibility of the concepts and mechanisms defining a contributory distributed system. Thanks to the two properties modularity and extendability, it is possible to implement easily different mechanisms in the prototype and incorporate easily changes and improvements on existing ones. The language selected for the implementation of the prototype is Java because it can handle heterogeneous nodes thanks to its Java Virtual Machine. Additionally, it's an object-oriented language, which naturally leans toward modularity. The prototype lays over Freepastry [29] which is an open-source implementation of Pastry [13] intended for deployment in the Internet. Pastry is an overlay network which allow us to create a decentralized distributed system and offers a set of functionalities to ensure communication between the nodes, handling the joining and leaving of nodes, creation of routing tables for each node, failure tolerance. Freepastry offers also an implementation of PAST's DHT [8] to handle a distributed storage system. All these elements concerning the current sate of the prototype and the design choices that have been made for CoDeS, will be considered as boundaries that delimit the possible solutions that could be proposed as a DSS for CoDeS.

1.3. Objectives and Approach

The main goal of this research is to provide a DSS for CoDeS. The aim of the state of the art that we've done for this research is to help us to find an existing solution that could correspond to our needs or, at least, some mechanisms that we could use to propose our own solution for a DSS.

The DSS that we intend to integrate must cope with the requirements of a contributory distributed system and in the following sections we've presented a lot of existing solutions that use mechanisms that correspond to these requirements. Mechanisms that ensure features as scalability, consistency, availability, fault tolerance, security and self-organization. In the next sections, we've presented also a decomposition of a DSS in layers that contains the different components constituting the system. We've described which elements from the taxonomy we've selected and which mechanisms from the survey could be implemented in our distributed storage system.

1.4. Contributions

The main contributions of this research can be summarized in the following categories.

State of the art of the existing distributed storage system

In the state of the art, we've studied the existing storage systems and we've depicted them using a taxonomy (describing

the different elements composing a DSS) to be able to analyze each element and see how they handle each feature. For each element of the taxonomy, we provide a definition and which solutions implement it. This way, we were able to separate the solutions suitable for our research and the ones that aren't.

Test and evaluation of freepastry's DHT

Thanks to the state of the art, we've been able to conclude that PAST is the best existing solution that could be used or adapted for CoDeS. To be sure that Freepastry's [29] implementation of PAST is the right answer for our needs , we've designed a series of tests that has helped us evaluating how Freepastry handles each one of the functional and non functional requirements and if it can be integrated without modifications in CoDeS or additional modifications or developments are needed in order to do so.

1.5. Outline

The rest of the document is organized as following: section 2 presents some works related to DSS and introduces the taxonomy used to select the best solution for this research. Section 3 presents the selected solution and describes the tests and evaluations that we've done to prove the validity of the suggestion. Finally, section 4 concludes this paper.

2. Distributed Storage System for CoDeS

2.1. Overview

In this section, we discuss studies related to DSSs. To be able to study the existing solutions in a detailed manner, we've based our study in a taxonomy that present each element constituting a distributed storage system and each feature characterizing it. For each concept, we've presented the solutions that implement it. This way, we were able to split the complexity of a DSS into small elements and analyzed each one of theme to select the best solution according to our needs and requirements.

2.2. Related Work

The work of [20] presents a taxonomy for distributed storage systems. This taxonomy finds distributed storage systems to offer a wide array of functionalities, employ architectures with varying degrees of centralization and operate across environments with varying trust and scalability. Furthermore, taxonomies on automatic management, federation, consistency and routing provide an insight into challenges faced by DSSs and the research to overcome them. The paper continues by providing a survey of distributed storage systems which exemplify topics covered in the taxonomy. Another interesting taxonomy is the one presented by [21]. The authors discuss the key concepts behind Data Grids and compare them with other data sharing and distribution paradigms such as content delivery networks, peer-to-peer networks and distributed databases. They also provide comprehensive taxonomies that cover various aspects of architecture, data transportation, data replication and resource allocation and scheduling. Finally, they map the proposed taxonomy to various Data Grid systems not only to validate the taxonomy but also to identify areas for future exploration.

2.3. Taxonomy

The taxonomy presented in this document is based on the two taxonomies described in [20] and [21]. The first taxonomy introduces several topics related to DSSs and it classifies them according to the following criterion: Function, Storage Architecture, Operating Environment, Usage Patterns, Consistency, Security, Autonomic management, Federation, Routing and Network Overlay. The second taxonomy is split into four sub-taxonomies. The first sub-taxonomy is from the point of view of organization. The second one deals with the transport technologies used within the distributed system. This not only covers well-known file transfer protocols but also includes other means of managing data transportation. A scalable, robust and intelligent replication mechanism is crucial to the smooth operation of a distributed system and the third sub-taxonomy presented takes into account concerns of distributed system environments such as meta-data and the nature of data transfer mechanisms used. The last sub-taxonomy categorizes resource allocation and scheduling research and looks into issues such as locality of data.

We found the three first sub-taxonomies presented by [21], to be very pertinent as a description of the different components of a DSS and they could be seen as a layered architecture. On one hand, by separating the different elements, we devise the complexity of the system and we can concentrate on specific topics. On the other hand, we can see how components can interact and collaborate with each other. Our taxonomy is based on these three sub-taxonomies and we've added topics from the first taxonomy that are absent from the second. These topics were added into their corresponding sub-taxonomy.

2.3.1. Organization

Figure 1 shows a taxonomy based on the various organizational characteristics of DSSs. These characteristics are present in different ways in different systems.

2.3.1.1. Function

In this section, we'll discuss application functional requirements of a DSS. The different functions that a DSS could offer are the following: Archival, General Purpose File System, Publish/Share, Performance and Custom.

Archival are meant to provide the end user with the ability to store and retrieve data. Thus it ensures consistent non volatile storage by achieving reliability even in the presence of failure. Achieving reliability, even in the event of failure, supersedes all other objectives and data replication is a key instrument in achieving this. Systems in this category are rarely required to make updates, their workloads follow a write-once and read-many pattern. Updates to an item are made possible by removing the old item and creating a new item and whilst this may seem inefficient, it is adequate for the expected workload. Examples of storage systems in this category include PAST [8] and CFS [2].

General Purpose File System is used to provide users with a file system like interface to achieve persistent non volatile storage. The applications access data stored as if it were in the local system. Most of systems based on this kind of systems are based on the POSIX API standards allowing existing applications to utilize storage without the need for modification or a re-build. Systems which fall into this category include Ivy [11], Pastis [23], Farsite [1], GPFS [15] and Coda [14].

Publish/share are more destined for publish and share operation where persistence is not as much important as the popularity of a document. There are two types of publish/share storage systems, the one which offer anonymity and the one which ensure file sharing. In the case of the first one, the most important aspect is the anonymity of the document and the person who published it, examples of systems which fall into this category include Free Haven [7] and Freenet [5]. Concerning the second one is to provide the capability to share files amongst users, examples of systems that offer this functionality are Napster [24] and Gnutella [25], these later are not present in our survey because they are not relevant for our research needs.

Performance is more aimed for distributed systems that require high performance access and retrieval of data. The majority of systems falling in this category follow the specifications recommended by parallel file systems. Nodes in this system are interconnected by a high bandwidth and stripe data to aggregate bandwidth. Systems which fall into this category include PVFS [3], GPFS [15] and Frangipani [17].

Custom systems possess a unique set of functional requirements and may be elaborated by combining the above systems. Examples of systems that may fit in this category are OceanStore [9], Freeloader [18] and Bayou [6].

Organization	
Function	Archival
	General Purpose FS
	Publish/Share
	Performance
	Custom
Storage Architecture	Client/Server
	Peer-To-Peer
	Custom
Scope	Inter-domain
	Intra-domain
Operating Environment	Trusted
	Partially Trusted
	Untrusted
Usage Patterns	Application
	User/Application
	User
Management	Managed
	Self Configure
	Self Optimize
	Self Protect
	Self Heal
Virtual Organization	Collaborative
	Regulated
	Economic
	Reputation-based

Figure 1: Organization

2.3.1.2. Storage Architecture

In this section we'll concentrate our attention into the organization of nodes into a DSS and how this architecture may influence behavior and functionality. We can split this topic in two main categories which are client-server and peer-to-peer architectures. In the first one, node can play the role of servers or clients, they can't be both. Whereas, nodes into a peer-to-peer architecture can play at the same time the role of a server and a client.

In a client/server architecture, the server is responsible for providing different services to clients. The advantage of this architecture is that the only nodes responsible for administration, management and optimizations are the server nodes. The servers are in charge of security, consistency, backup and responding to clients requests which could ensure a good level of performance for a few numbers of nodes. However, for a DSS aimed for millions, servers constitute a single point of

failure.

There are two types of client/server architecture, globally centralized and locally centralized. The former use one server as the principal node in charge of providing services and servicing requesting clients, the later is based on a set of servers to alleviate the overload on the unique server. Locally centralized architecture scale better and achieve availability by replicating data among the set of servers. However, the system still have the limits of a client/server architecture giving that it's based on limited set of servers which are responsible for the whole distributed storage system functionalities. Examples of systems that fall into this category include PVFS [3], Bayou [6] and Coda [14].

As a response to the limits of a client/server architecture appeared a new paradigm named peer-to-peer. Peer-to-peer architectures are more suitable for untrusted environments, highly dynamic and constituted by thousands of distributed machines. In this architecture a node could play the role of a server, a client or both at the same time. The advantages offered by peer-to-peer systems are high scalability, self-organization and availability. However, they are more exposed to more threats from misbehaving nodes compared with a controlled client/server environment, administration and management of this kind of architecture is more complex and the response time for clients' requests could be slower if the mechanisms used for localization, replication and consistency are not optimized.

There are three main categories of peer-to-peer architectures, Globally centralized, Locally centralized, and Pure Peer-to-peer. The Globally centralized architecture uses a central server for data localization, when a user needs to locate a certain data, she requests this information from the central server, once she get the location of the node which contain the data, she retrieve it directly from the node. The inconvenient in this case is the poor level of scalability and that the server represents a Single Point of Failure. Examples of systems that fall into this category include Napster [24]. Locally centralized architecture emerged in response to the limits of Globally centralized architectures, they use a set of nodes with high performance called super nodes which play the role of severs. The main functionality of these servers is to provide clients with information related to data location and availability. The use of a set of servers alleviates some of the limits of a single server but are not optimized for a large scale dynamic distributed system. Examples of systems that fall into this category include OceanStore [9], Farsite [1], CFS [2], GPFS [15] and Frangipani [17]. Finally, pure peer-to-peer architecture doesn't dispose of specific servers, each node could play the role of a server which makes peer-to-peer systems very adaptable in a highly dynamic environment where node can join and leave at will. As we've already said, the advantages of pure peer-to-peer are related to its capacity for adaptability and to be optimized for large scale distributed systems. However, there are few points that make the design and implementation of a peer-to-peer system very complex. The nodes participating in this kind of systems doesn't have the same level of performance, storage capacity and bandwidth, therefore peer-to-peer systems need to have mechanisms that take into account this asymmetry in the network. Security mechanisms as authentication and authorization need to be applied in order to reduce the number of threats that the system is exposed to. Self-organization is another important point that needs to be taken into account when designing a pure peer-to-peer system to reduce complexity of the system. Examples of systems that fall into this category include Ivy [11], Pastis [23], PAST [8], Freenet [5] and Free Haven [7].

As we've seen the choice of the architecture has a major impact on the system functionalities and how it could manage the different features offered as scalability, consistency and security. Client/Server architectures are more suited for controlled environments, where scalability is not required and the important feature is quality of service. In the contrary, peer-to-peer architectures are more suited for highly dynamic environments and very large scale distributed storage systems where scalability and security are very important.

2.3.1.3. Scope

The scope of a distributed storage system can correspond to a limited set of nodes in a single domain (intra-domain) where the environment is more controlled and security concerns are restricted inside the boundaries of the domain. Or it can be aimed at a very large environment constituted by several domains (inter-domain), this time the system is often relied on untrusted nodes and it's highly dynamic. Examples of systems that fall into intra-domain category include Frangipani [17], Freeloader [18] and bayou [6] and examples of systems that fall into inter-domain category include Ivy [11], Pastis [23], PAST [8], OceanStore [9], Farsite [1], Freenet [5], CFS [2] and Free Haven [7].

2.3.1.4. Operating Environment

In this section we will discuss the environment that the distributed storage system will operate in. This section concentrates more specifically in security aspects and how the reputation of the nodes are seen by the system in order to add specific mechanisms to ensure security in all possible environments. There is three main types of environments: Trusted, Partially Trusted and Untrusted.

Trusted environments are mostly controlled and restricted environment where all the nodes are trusted. Administration tasks are limited to the boundaries of the domain and they are simpler than the ones used in large scale systems with untrusted nodes. As the system is very controlled there is no need to worry about unpredictability which alleviate the workload required for an untrusted environment. Examples of systems that fall into this category include Frangipani [17], Freeloader [18].

Partially Trusted environments are formed by trusted and untrusted nodes. They are limited by the boundaries of an organization and users are only personnel within the organization. These restrictions limit the types of attacks that threaten the environment to the ones coming from misbehaving personnel or someone who got access to a machine inside the organization. The system needs to offer mechanisms to avoid these kinds of misbehaviors to ensure a high level of security. In this category of environments, the network is a shared resource and the DSS needs to be able to work in harmony with other systems, avoiding interference and conflicts with them. Examples of systems that fall into this category include Ivy [11], Farsite [1] and Coda [14].

Untrusted environment are more suited for large scale systems where all nodes are untrusted and where network infrastructure is shared and open to everyone. In this environment nodes behavior is very unpredictable thus the impact on workload which explains the presence of failures. A huge number of attacks threaten this type of environments and mechanisms need to be developed in order to minimize the threats and ensure a good level of security. Examples of systems that fall into this category include Pastis [23], PAST [8], OceanStore [9], Freenet [5], CFS [2] and Free Haven [7].

Operating environment has a major influence on system design and the predictability of workload. Consequently, it must be taken into account in the design of a DSS and the mechanisms offered by this later.

2.3.1.5. Management

The management of distributed storage systems can be done automatically by mechanisms designed to adapt the system to the dynamism and unpredictability of the environment or can be managed manually by administrators. The complexity of large distributed storage system require the intervention of administrators to ensure the stability of the system by doing tasks such as resource monitoring, user authorization and data replication. However, research is leading to autonomic or self-organizing, self-governing systems. Examples of systems that fall into the managed category include Farsite [1] and Frangipani [17]. Examples of systems that fall into the autonomic category include Pastis [23], PAST [8], OceanStore [9], Freenet [5], CFS [2], Free Haven [7], GPFS [15] and Coda [14].

2.3.1.6. Virtual Organization

A Virtual Organization is formed when different organizations pool resources and collaborate in order to achieve a common goal. A VO defines the resources available for the participants and the rules for accessing and using the resources and the conditions under which the resources may be used. DSS are formed by VOs and therefore the design of VOs reflects the objectives of the distributed system.

A collaborative system is when resources are aggregated for a single goal. Users share their unused resources to allow a better exploitation of the distributed system in order to achieve the goal of the collaborative community. Examples of systems that fall into this category include Ivy [11], Pastis [23], PAST [8], Farsite [1], Freenet [5], CFS [2], Frangipani [17], Freeloader [18], Coda [14] and Bayou [6]. In systems more controlled, regulated systems are more suited where a single organization defines rules for accessing and sharing resources. Examples of systems that fall into this category include GPFS [15]. Another type of organization is based on economics principles where providers make available resource for end users in exchange of fees. Examples of systems that fall into this category include OceanStore [9]. A reputation based distributed system corresponds to a system based on a reputation mechanism that allow the system to know which machines to trust and to which it can give more privileges. Examples of systems that fall into this category include Free Haven [7].

2.3.2. Data Transport

The data transport mechanism is one of the fundamental technologies underlying a distributed system. Data transport involves not just movement of bits across resources but also other aspects of data access such as security, access controls and management of data transfers. A taxonomy for data transport mechanisms is shown in Figure 2.

2.3.2.1. Function

Data transport in distributed storage system can be modeled as a three-tier architecture: Transfer protocol, Overlay

network and File I/O mechanism.

3.3.2.4.1. Transfer protocol

The bottom layer corresponds to the one responsible of the exchange of data between two nodes, it takes care of simple bit movement between two hosts on a network. These protocols are common languages used by two nodes in a network to initiate and control data transfers. Examples of protocols that fall into this category include FTP, GridFTP, TCP (PVFS [3]), TCP/UDP (PAST [8], Freenet [5], FreeLoader [18]) and RPC (Coda [14]).

3.3.2.4.2. Overlay Network

The second tier is the overlay network which adds specific functionalities to applications over the Internet protocol to respond to specific needs. It's mainly used for locating and routing data between different nodes of the distributed system. It provides services such as storage in the network, caching of data transfers for better reliability and the ability for applications to manage transfer of large data. Examples of systems that fall into this category include Pastry [13], Tapestry [19], Chord [16] and CAN [27].

3.3.2.4.3. File I/O mechanism

A File I/O mechanism allows an application to access remote files as if they are locally available. This mechanism presents to the application a transparent interface through APIs that hide the complexity and the unreliability of the networks. Systems using this feature include Ivy [11], Pastis [23], Farsite [1], GPFS [15] and Coda [14].

Data Transport		
Function	File I/O mechanism	
	Overlay Network	
	Transfer Protocol	
Security	Authentication	Passwords
		Cryptographic keys
	Authorization	Coarse-grained
		Fine-grained
	Encryption	SSL
		Unencrypted
		Encrypted
	ACL	
	Reputation	
P2P Network Overlay	Node ID Assignment	
	Routing Table Maintenance	
	Secure Message Forwarding	
	Byzantine Agreement	
	Onion routing	
	Probabilistic Routing	
Fault tolerance	Fault tolerance	
	Restart Transmission	
	Resume Transmission	
	Cached Transfers	
Transfer Mode	Block	
	Stream	
	Compressed	
	Bulk Transfers	

Figure 2: Data Transport

2.3.2.2. Security

In a DSS, security is a very important requirement which goal is to ensure proper authentication, file integrity and confidentiality. Security in DSSs can be divided into three main categories: authentication, authorization and encryption of data. Authentication can be based on either passwords or symmetric or asymmetric public key cryptographic protocols. The second category is authorization which is used to limit the access of users to specific directories or files. This category could be split into two sub categories: Coarse-grained and fine-grained. The first one uses methods such as UNIX file permissions to restrict the number of files that are accessible to the user. The second one is more restrictive even for

authorized users, it's meant for applications that have strict controls on the distribution of data. Fine-grained access control methods that may be employed to achieve these requirements include time- and usage-limited tickets, Access Control Lists (ACLs), Role Based Access Control (RBAC) methods and Task-Based Authorization Controls (TBAC). Finally, during data transfers encryption could be used. The most known mechanism of data encryption is through SSL (Secure Sockets Layer).

Security	Ivy	PAST	PASTIS
Cryptographic keys	x	x	x
ACL			x
Reputation	x		

2.3.2.3. Fault-tolerance

The system must be capable of handling machines disconnection and systems failures by keeping a stable state of the system and without the knowledge of end users, in other words these failures need to be transparent for the user.

2.3.2.4. Transfer mode

The last category is the transfer mode that can be used in the DSS. Traditional modes that could be supported by the mechanism are: Block, Stream and Compressed. However in the case of large data transfer, there could be some restrictions related to protocols such as FTP and TCP which were initially designed for low bandwidth, high latency networks. Therefore, several optimizations have been suggested for improving the performance of data transfers in distributed environments by reducing latency and increasing transfer speed. Some of them are listed below:

- Parallel data transfer: is the ability to use multiple data streams over the same channel to transfer a file. This also saturates available bandwidth in a channel while completing transfer.
- Striped data transfer: is the ability to use multiple data streams to simultaneously access different blocks of a file that is partitioned among multiple storage nodes (also called striping). This distributes the access load among the nodes and also improves bandwidth utilization.
- Auto-resizing of buffers: is the ability to automatically resize sender and receiver TCP window and buffer sizes so that the available bandwidth can be more effectively utilized.
- Container operations: is the ability to aggregate multiple files into one large dataset that can be transferred or stored more efficiently. The efficiency gains come from reducing the number of connections required to transfer the data and also, by reducing the initial latency.

These enhancements are grouped under bulk transfer mode.

2.3.3. Data Replication and Storage

The main goal of a DSS is to offer to users means to exploit unused distributed disk space and to store and retrieve data in a more flexible way than in the case of one single machine. These features are available thanks to the concept of replication which ensure scalability of the collaboration, reliability of data access and perseverance of bandwidth.

The replica manager is in charge of the creation and management of replicas according to the demands of the users and the availability of storage, and a catalog or a directory keeps track of the replicas and their locations. Applications can request the catalog to know the number and the locations of available replicas of a specific data. The manager and the catalog can be split in two different entities or can be merged into one entity. The replication mechanism can be triggered as a response to user requesting replication or as response to the manager decision to replicate some data.

The important elements of a replication mechanism are therefore the architecture of the system and the strategy followed for replication.

Data Replication		
Model (Architecture)	Centralized	
	Decentralized	
Topology (Architecture)	Hierarchical	
	Flat	
	Hybrid	
Storage Integration (Architecture)	Tightly-coupled	
	Loosely-coupled	
	Intermediate-coupled	
Transfer Protocols (Architecture)	Open protocols	
	Closed protocols	
Meta-data (Architecture)	Attributes	System
		User-defined
	Update type	Active
		Passive
Update Propagation (Architecture)	Synchronous	
	Asynchronous	Epidemic
		On-demand
		Periodically
Catalog Organization (Architecture)	Tree	
	Hash-based	
	DBMS	
Method (Strategy)	Static	
	Dynamic	
Granularity (Strategy)	Container	
	Dataset	
	File	
	Fragment	
Objective Function (Strategy)	Locality	
	Popularity	
	Update costs	
	Economic	
	Preservation	
	Publication	
Consistency	Optimistic	
	Strong	
Conflict resolver		
Availability		
Scalability		
Reliability		

Figure 3: Data Replication

2.3.3.1. Data Replication Architecture

3.3.2.4.1. Model

The model chosen for a distributed storage system will determine the organization of the nodes in the network and how replication will be implemented in the system. A centralized system for example will have a centralized server in charge of replication which will be the first one to be updated and it will propagate the updates to the other nodes. Examples of systems that fall into this category include OceanStore [9], PVFS [3] and Coda [14]. In the case of a decentralized system, replicas will be distributed over different nodes of the network which will need synchronization mechanisms to ensure that all replicas are up to date. Examples of systems that fall into this category include Ivy [11], Pastis [23], PAST [8], Farsite [1], Freenet [5], CFS [2], Free haven [7] and Freeloader [18].

3.3.2.4.2. Topology

Topology is what defines how the nodes will be organized, we can distinguish three groups: Hierarchy, Flat and Hybrid.

Hierarchical topologies have tree-like structure in which updates propagate through definite paths. Examples of systems that fall into this category include OceanStore [9], Farsite [1] and PVFS [3]. Flat topologies are found within P2P systems and progression of updates is entirely dependent on the arrangements between the peers. These can be both structured and unstructured. Examples of systems that fall into this category include Ivy [11], Pastis [23], PAST [8], Freenet [5] and CFS [2]. Hybrid topologies can be achieved in situations such as a hierarchy with peer connections at different levels.

3.3.2.4.3. Storage Integration

The relation between replication and storage is very important and has a direct impact on scalability, robustness, adaptability and applicability of the replication mechanism. Tightly-coupled replication mechanisms are tied to the storage architecture on which they are implemented. The replication system controls the file system and I/O mechanism of the local disk. Such systems more or less try to behave as a distributed file system such as NFS (Network File System) as they aim to provide transparent access to remote files to applications. Intermediately-coupled replication systems have control over the replication mechanism but not over the storage resources. The file systems are hosted on diverse storage architectures and are controlled by their respective systems. However, the replication is still initiated and managed by the mechanism, and therefore it interacts with the storage system at a very low-level. In the previous cases replication is executed in a transparent way to users and applications, however theses later could be responsible for replication, in this case we talk about Loosely-coupled replication mechanisms. The mechanism exerts no control over the file system. Such mechanisms interact with the storage systems through standard file transfer protocols and at a high level.

3.3.2.4.4. Transfer protocol

In this paragraph we'll talk about the data transport protocols used within replica management systems. There is two types of transfer protocols: open and closed. The former ones are meant for systems where users and applications can replicate data independently from the replication management system. However, in this case users and applications have to update the replicas locations in the catalog as they transferred data outside the replication management system. In the case of closed protocols access to the replicas is restricted to their client libraries.

3.3.2.4.5. Metadata

Metadata contains information related to the data stored all over the distributed storage system. This way users can query for data using attributes that are more familiar to them. Metadata can have two types of attributes: one is system-dependent metadata, which consists of file attributes such as creation date, size on disk, physical location and file checksum and the other is user-defined attributes which consist of properties that depend on specific applications. There is two ways to update metadata: actively and passively. The first one is achieved by the replica management system. The second one is triggered by the users when they create new replicas, modify existing ones or add a new file to the catalog. Examples of systems using this feature Ivy [11] and Pastis [23].

3.3.2.4.6. Replica Update Propagation

Generally, data is updated in one location and these updates are propagated afterward to the rest of replicas. There is two modes for replica update propagation: synchronous or asynchronous mode. The advantage of the synchronous mode is that the probability of having out of date replicas is very low as all the replicas are updates just after one the replica has been updated, however the disadvantage is that this mechanism can be expensive as it can take long time to update all the replicas. Asynchronous updating can be epidemic, the primary copy is changed and the updates are propagated to all the other replicas or it can be on-demand wherein replica sites subscribe to update notifications at the primary site and decide themselves when to update their copies.

3.3.2.4.7. Catalog Organization

Catalogs can be organized as trees, they can be organized on the basis of document hashes as has been the case in P2P networks or finally by storing the catalogs within a database as in the case of RSB. Examples of systems using this feature Ivy [11], Pastis [23] and CFS [2].

2.3.3.2. Replication Strategy

Replication strategies determine when and where to create a replica of the data. These strategies are guided by factors such as demand for data, network conditions and cost of transfer.

3.3.2.4.1. Method

This section refers to whether the strategies are static or dynamic. In dynamic strategies, decisions are made according to changes in demand and bandwidth and storage availability however it induce overhead due to larger number of operations that they undertake as these are run at regular intervals or in response to events. Dynamic strategies are able to recover from failures such as network partitioning. As the performance in dynamic strategy could be influenced by the overload of the mechanism to ensure replication, it is more advisable to use a static strategy if the resource conditions are fairly stable in a distributed system over a long time.

3.3.2.4.2. Granularity

Granularity corresponds to the level of subdivision of data that the strategy works with. In the case of replication of multiple files at the same time, we talk about the granularity of datasets. Another level of granularity is the one dealing with individual files. Finally, some strategies deal with smaller subdivisions of files such as objects or fragments.

3.3.2.4.3. Objective Function

Possible objectives of a replication strategy are to maximize the locality or move data to the point of computation, to exploit popularity by replicating the most requested datasets, to minimize the update costs or to maximize some economic objective such as profits gained by a particular site for hosting a particular dataset versus the expense of leasing the dataset from some other site. Preservation driven strategies provide protection of data even in the case of failures. Another possible objective function for a replication strategy is to ensure effective publication by propagating new files to interested clients.

3.3.2.4.4. Consistency

This section discusses various mechanisms employed by DSSs to ensure data remains consistent even in the presence of events which challenge it such as concurrent updates. There two types of consistency: strong and optimistic. The primary aim of strong consistency is to ensure data is viewed and always remains in a consistent state. To maintain strong consistency, locking mechanisms need to be employed, it must also respect ACID (Atomic, Consistency, Isolation and Durability) principles. In the case of optimistic consistency, the primary purpose is to keep data consistent without imposing the restrictions associated with strong consistency. Optimistic consistency allows multiple readers and writers to work on data without the need for a central locking mechanism. Examples of systems using this feature Ivy [11] and Pastis [23].

3.3.2.4.5. Conflict Resolver

In DSS the data is shared between all the participants in the network, therefore conflicts caused by concurrent update may occur, especially in multiuser read/write systems. The system must offer mechanisms that can, on one hand prevent this kind of problems and on the other hand be in charge of detecting the presence of conflicts and resolve them to ensure a stable state of the data in the system. Examples of systems using this feature Ivy [11] and Pastis [23].

3.3.2.4.6. Availability

Even if the data is shared among several machines, the user must see it as if it were in its own machine, thus it must offer simple interfaces for end user applications to access data and services in the same way as in the case of a single machine. Examples of systems using this feature PAST [8] and CFS [2].

3.3.2.4.7. Scalability

When we talk about distributed systems, we talk about thousands of interconnected machines all over the world, these machines can join and leave the system whenever they want. The system must not only ensure a good degree of performance in the case of a limited number of machines but must maintain this degree of performance in the case of the joining of an increased number of machines.

2.4. Analysis of existing solutions

We'll start with the existing DSSs that don't correspond to a contributory distributed system and are meant for other specific area and requirements.

2.4.1. OceanStore

The first solution is OceanStore [9] which is a utility infrastructure designed to span the globe and provide continuous access to persistent information. Such a utility is highly-available from anywhere in the network, employ automatic replication for disaster recovery, use strong security by default, and provide performance that is similar to that of existing LAN-based networked storage systems under many circumstances. However, the main goal of OceanStore is to offers all these features to customers in exchanges of a monthly fee. Additionally, OceanStore architecture is based on dedicated servers able to participate in protocols for distributed consistency management. OceanStore consider that most of the servers are working correctly most of the time.

2.4.2. Farsite

Farsite [1] is a secure, scalable file system that logically functions as a centralized file server but is physically distributed among a set of untrusted computers. Farsite provides file availability and reliability through randomized replicated storage; it ensures the secrecy of file contents with cryptographic techniques. Farsite’s intended workload and its expected machine characteristics are those typically observed on desktop machines in academic and corporate settings. These workloads exhibit high access locality, a low persistent update rate, and a pattern of read/write sharing that is usually sequential and rarely concurrent. Even if Farsite doesn't require a lot of manual administration of the distributed system, it needs administrators intervention to authenticate new users and machines as they join the system. Finally, it does not efficiently support large-scale write sharing of files.

2.4.3. PVFS and GPFS

PVFS [3] is a parallel file system for Linux clusters. It's intended both as a high-performance parallel file system that anyone can download and use and as a tool for pursuing further research in parallel I/O and parallel file systems for Linux clusters. This solution doesn't correspond to a contributory distributed system because it needs a dedicated set of servers that will constitute the cluster, furthermore It must provide high bandwidth for concurrent read/write operations from multiple processes or threads to a common file. Another example of this kind of systems is GPFS [15] which is a parallel file system for cluster computers that provides, as closely as possible, the behavior of a general-purpose POSIX file system running on a single machine. It's meant to be used on supercomputers.

2.4.4. Freenet and Free Haven

Freenet [5] is a peer-to-peer network application that permits the publication replication and retrieval of data while protecting the anonymity of both authors and readers. Files are referred to in a location independent manner and are dynamically replicated in locations near requestors and deleted from locations where there is no interest. Another example that falls into this category is Free Haven [7]. In these two solutions the most important aspects are anonymity and the publication are based on the popularity of the shared data.

2.4.5. Frangipani

Frangipani [17] is a new scalable distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage. It's highly available in spite of component failures. It requires minimal human administration, and administration doesn't become more complex as more components are added. However, it's meant to run in a cluster of machines that are under a common administration and can communicate securely which don't correspond to a contributory system.

2.4.6. FreeLoader

FreeLoader [18] aggregates unused desktop storage space and I/O bandwidth into a shared cache/scratch space, for hosting large, immutable datasets and exploiting data access locality. Freeloader is based on a secure environment and no malicious intent in the workstation users' part. The datasets are large, immutable files (write–once–read–many) ranging from several hundred megabytes to several gigabytes. Datasets are almost always held at a remote primary source (which is rarely modified) while users analyze their local copies. The primary sources are dedicated servers in the Freeloader network. This system is more meant for data intensive storage for scientists.

2.4.7. Coda

Coda [14] is a distributed file system that is resilient to failures that typically occur in a workstation environment. It provides high availability through the use of two distinct but complementary mechanisms, server replication and

disconnected operation. It's based on client/server architecture where servers are physically secure, run trusted system software and are monitored by operational staff.

2.4.8. CFS

The Cooperative File System (CFS) [2] is a peer-to-peer read only storage system that provides provable guarantees for the efficiency, robustness, and load-balance of file storage and retrieval. CFS does this with a completely decentralized architecture that can scale to large systems. CFS servers provide a distributed hash table (DHash) for block storage. CFS clients interpret DHash blocks as a file system. CFS present interesting features that could be used in a DSS for CoDeS, however, it's meant for read only storage. It will be interesting to do further studies concerning this system to see if it's possible to adapt it to a multi user read/write storage system.

2.5. Selected solutions

In this section we'll discuss the existing solutions that share common aspects with a collaborative distributed system. With the help of some experimentation and further studies they have the potential to be adapted for a DSS running as a service in CoDeS.

2.5.1. PAST

PAST [8] share a lot of features with the DSS needed for CoDeS. It's a decentralized system which handles availability, scalability, persistence, self-organization and security. An important feature offered by PAST that could be used in CoDeS is the ability of selecting potential nodes that will store replicas and it do this taking into account a lot of variables as geographic location, ownership, administration, network connectivity and rule of law. The routing scheme used by PAST is called Pastry [13] and it's a complete and efficient mechanism to reliably route requests to the appropriate nodes.

However, PAST use a quota system to control users' contribution and resource use which doesn't correspond to CoDeS philosophy. Additionally, PAST may optionally use third party components called brokers which control how much storage must be contributed and/or may be used by users which also don't correspond to CoDeS requirements.

2.5.2. Pastis

Pastis [23] is a good example of a storage system that has been designed based on existing solution as PAST and routing mechanism as Pastry and that has been adapted to specific needs. For our distributed storage system, it could be interesting to not use an existing solution or propose a new one but adapt an existing one to our specific requirements.

Pastis introduces mechanisms to handle consistency. On one hand there is the close-to-open model which is implemented by retrieving the latest inode from network when the file is opened and keeping a cached copy until the file is closed. Any following read requests are satisfied using the cached inode. On the other hand it presents a read-your-writes model guarantees that when an application opens a file, the version of the file that it reads is not older than the version it previously wrote. Once the file is opened, file updates are performed as in the close-to-open model.

Pastis ensures write access control and data integrity through the use of standard cryptographic techniques and ACL certificates. Pastis does not currently provide read access control, but users may encrypt files' contents to ensure data confidentiality if needed.

2.5.3. Ivy

Ivy [11] is a decentralize system offering a multi/user read/write peer-to-peer file system. It uses DHash to implement a distributed storage system; it's based only on a set of logs, one log per participant which resolves problems as locking. It handles security by using cryptography and allowing users to choose which other nodes to trust.

All these characteristics match the requirements of the storage system needed for CoDeS. However, aspects as write-read consistency and replication are not implemented in the current version of Ivy and if we want to use it for CoDeS, we need to do a study on how we can implement these features (important for our research) without impacting Ivy's performance. Additionally, we need to see how we can optimize Ivy to handle a big number of cooperating participants.

3. Freepastry's implementation of PAST

3.1. Overview

In the state of the art, we have studied the existing distributed storage systems and we've classified them according to a taxonomy to be able to identify the features, the requirements and objectives of each solution. This way, we've been able to identify the solutions that weren't suitable for our research and the ones that could be used or adapted to cope with CoDeS requirements. The selected solutions are PAST, PASTIS and Ivy. In the previous section, we've explained the reasons behind these choices. Now we'll present the solution that we consider the most suitable for CoDeS.

To make a suggestion, we needed to take into account the current state of the prototype, the solutions that we've select in the previous section and the boundaries defined by the set of requirements. The three selected solutions use a DHT to ensure the storage of data in the distributed system. PAST and Pastis are based on Pastry as an overlay network that ensures communication and transport between the nodes constituting the distributed system. Additionally, we've said that the prototype of CoDeS relays on Freepastry [29] to create a virtual ring that will contain the different nodes and it's capable of managing communication, routing and transport. Giving that Freepastry offers a library that handles the management of a DHT, it seems logic to us to consider PAST the more adequate between the three solutions selected.

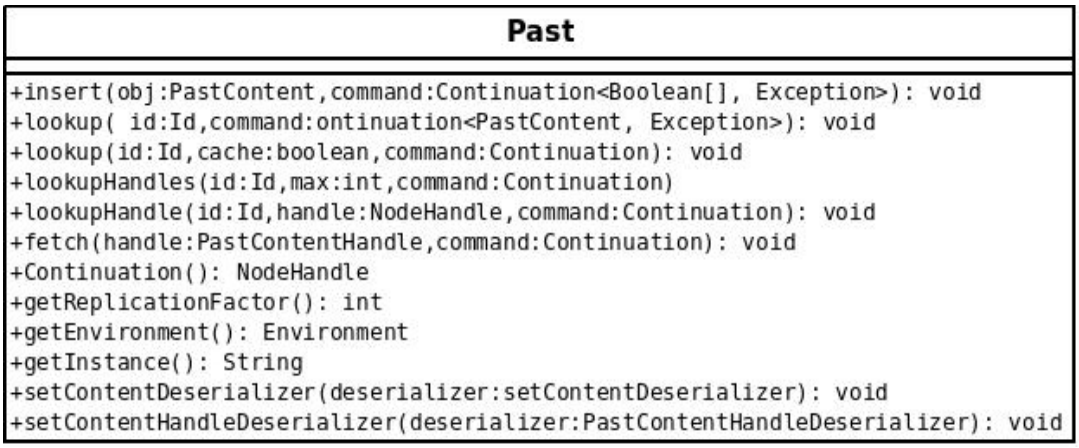
3.2. Freepastry's PAST architecture

In the next paragraphs, we'll describe the packages implemented in Freepastry [29] responsible of handling PAST's operations.

3.2.1. rice.p2p.past package

This package offers the core interfaces for handling PAST operations. In PAST interface for example, we can find the insert and lookup operations that allow the storing and retrieval of an object in the virtual ring. Thanks to Freepastry [29] implementation, we can ensure a complete decentralization of the storage system (we don't need specific nodes to be responsible of specific tasks). All nodes have the same capabilities to store and retrieve objects. Freepastry offers also an important feature which is scalability but we need to test it in the case when the system is used for storage. An important aspect to test is if the PastContent can be constituted of small and large files, this is very important if we want to respect the functional requirements.

3.2.1.1. Past interface



This interface is exported by all instances of Past. An instance of Past provides a distributed hash table (DHT) service. Each instance stores tuples consisting of a key and an object of a particular type, which must implement the interface PastContent. Past is event-driven, so all methods are asynchronous and receive their results using the command pattern.

3.2.1.2. Methods

insert *:* Inserts an object with the given ID into this instance of Past. Asynchronously returns a PastException to command, if the operation was unsuccessful. If the operation was successful, a Boolean[] is returned representing the responses from each of the replicas which inserted the object.

lookup *:* Retrieves the object stored in this instance of Past with the given ID. Asynchronously returns a PastContent object as the result to the provided Continuation, or a PastException. This method is provided for convenience; its effect is identical to a lookupHandles() and a subsequent fetch() to the handle that is nearest in the network. The client has the possibility to authenticate the object. In case of failure, an alternate replica of the object can be obtained via lookupHandles() and fetch(). This method is not safe if the object is immutable and storage nodes are not trusted. In this case, clients should use the lookUpHandles method to obtains the handles of all primary replicas and determine which replica is fresh in an application-specific manner. By default, this method attempts to cache the result locally for future use. Applications which do not desire this behavior should use the lookup(id, boolean, command) method.

lookupHandles *:* Retrieves the handles of up to max replicas of the object stored in this instance of Past with the given ID. Asynchronously returns an array of PastContentHandles as the result to the provided Continuation, or a PastException. Each replica handle is obtained from a different primary storage root for the the given key. If max exceeds the replication factor r of this Past instance, only r replicas are returned. This method will return a PastContentHandle[] array containing all of the handles. If we specify a specific handle, we can retrieve it for the given object stored on the requested node.

fetch *:* Retrieves the object associated with a given content handle. Asynchronously returns a PastContent object as the result to the provided Continuation, or a PastException. The client has the possibility to authenticate the object. In case of failure, an alternate replica can be obtained using a different handle obtained via lookupHandles().

3.2.1.3. PastContent interface

This interface must be implemented by all content objects stored in Past. The interface allows applications to control the semantics of an instance of Past. For instance, it allows applications to control which objects can be inserted (e.g., content-hash objects only), what happens when an object is inserted that already exists in Past, etc.

3.2.1.4. Methods

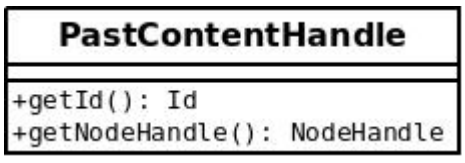
checkInsert : Checks if an insert operation should be allowed. Invoked when a Past node receives an insert request and it is a replica root for the id; invoked on the object to be inserted. This method determines the effect of an insert operation on an object that already exists, it computes the new value of the stored object, as a function of the new and the existing object.

getHandle : Produces a handle for this content object. The handle is retrieved and returned to the client as a result of the Past.lookupHandles() method.

getId : Returns the Id under which this object is stored in Past.

isMutable : States if this content object is mutable. Mutable objects are not subject to dynamic caching in Past.

3.2.1.5. PastContentHandle interface



This interface must be implemented by all content object handles. This interface represents an object that is stored on a specific replica. Thus, the validity of a handle is sometimes very short, and applications should keep references to an object by Id rather than handle.

3.2.1.6. Methods

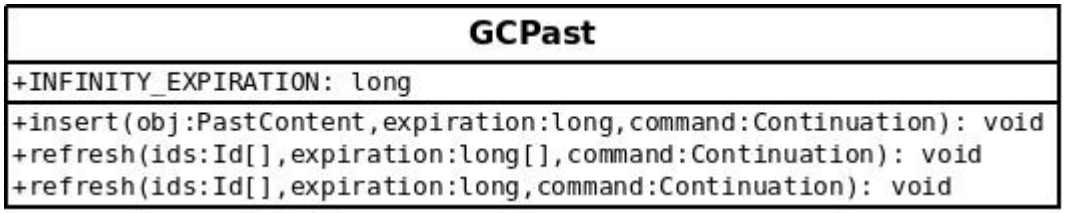
getId : get the id of the PastContent object associated with this handle

getNodeHandle : get the NodeHandle of the Past node on which the object associated with this handle is stored

3.2.2. package rice.p2p.past.gc

This package is used to manage the deletion of objects in the distributed storage system using timeout. This package will allow us to handle storage space efficiently.

3.2.2.1. GCPast interface



This interface represents an extension to the Past interface, which adds support for garbage collection. Individual objects are each given timeout times. The timeout times are used to determine when space can be reclaimed. Applications must periodically invoke the refresh() operation on all objects which they are interested in. Otherwise, objects may be reclaimed and will be no longer available.

3.2.2.2. Attributes and Methods

INFINITY_EXPIRATION : Timeout value which indicates that the object should never expire. Note that objects with this timeout value will be deleted in the year 292473178. If this is a problem, applications should check for this value explicitly.

insert : Inserts an object with the given ID into this instance of Past. Asynchronously returns a PastException to command, if the operation was unsuccessful. If the operation was successful, a Boolean[] is returned representing the responses from each of the replicas which inserted the object. This method is equivalent to insert(obj, INFINITY_EXPIRATION, command) as it inserts the object with a timeout value of infinity. This is done for simplicity, as well as backwards-compatibility for applications.

refresh : Updates the objects stored under the provided keys id to expire no earlier than the provided expiration time. Asyncroniously returns the result to the caller via the provided continuation. The result of this operation is an Object[], which is the same length as the input array of Ids. Each element in the array is either Boolean(true), representing that the refresh succeeded for the corresponding Id, or an Exception describing why the refresh failed. Specifically, the possible exceptions which can be returned are:

- ObjectNotFoundException : if no object was found under the given key
- RefreshFailedException : if the refresh operation failed for any other reason (the getMessage() will describe the failure)

3.2.3. package rice.p2p.replication

This package is responsible for object replication. It will allow us to have availability of objects giving that a storage of an object will be done in multiple nodes. Additionally, it will allow us to handle failure tolerance giving that if a node fails, we'll be able to retrieve the object from another node. These two feature are very important for our research and need to be tested carefully.

3.2.3.1. Replication interface

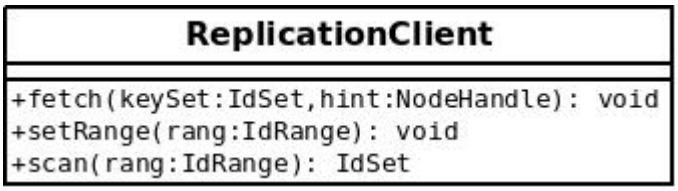
This interface is exported by Replication Manager (RM) for any applications which need to replicate objects across k+1

nodes closest to the object identifier in the NodeId space. The 'closest' (to the object identifier) of the k+1 nodes is referred to as the 0-root in which the object is stored by default when not using the replica manager. Additionally the RM assists in maintaining the invariant that the object is also stored in the other k nodes referred to as the i-roots (1<=i<=k). In the RM literature, k is called the ReplicaFactor and is used when an instance of the replica manager is being instantiated.

3.2.3.2. Method

replicate : Method which invokes the replication process. This should not normally be called by applications, as the Replication class itself periodically invokes this process. However, applications are allowed to use this method to initiate a replication request.

3.2.3.3. ReplicationClient interface



This interface should be implemented by all applications that interact with the Replica Manager.

3.2.3.4. Methods

fetch : This method is invoked to notify the application that it should fetch the corresponding keys in this set, since the node is now responsible for these keys also.

setRange : This method is used to notify the application of the range of keys for which it is responsible. The application might choose to react to call by calling a scan(complement of this range) to the persistence manager and get the keys for which it is not responsible and call delete on the persistence manager for those objects.

scan : This method should return the set of keys that the application currently stores in this range. Should return an empty IdSet (not null), in the case that no keys belong to this range.

3.2.3.5. ReplicationPolicy interface

This interface represents a policy for Replication, which is asked whenever the replication manager need to make an application-specific decision.

3.2.3.6. Method

difference : This method is given a list of local ids and a list of remote ids, and should return the list of remote ids which need to be fetched. Thus, this method should return the set B-A, where the result is a subset of B.

3.3. CoDeS Persistent Storage Module

CoDeS architecture is based on modules. Each module implements a part of the functionality of CoDeS, and they interact with each other in the local host to have access to their respective functionality. All the modules are present in every node executing CoDeS. Some of them provide their functionality by cooperating with the modules in other nodes using distributed mechanisms. To ensure extendibility, each module implement an interface that describes the behavior of the module. To use or interact with a module the other classes and modules only use the interface methods, this way the system will not relay on a specific implementation. This kind of architecture offers flexibility to try out different implementations and use the one that best fits our needs.

To integrate freepastry's PAST in CoDeS, we'll need to modify the PersistentStorageModule in order to implement PAST interface. This way, it will be able to use easily all the features offered by freepastry's PAST.

3.4. Evaluation

3.4.1. Test Environement

We've tested Freepastry's [29] DHT independently from CoDeS. This way we've been able to check which requirements it can handle and in case of problems we were sure that they are related to Freepastry's mechanisms and not the ones from CoDeS. Another advantage of this approach is that we've narrowed causes of problems and we've been able to concentrate on one issue at a time.

For the tests, we've used a virtual machine running Ubuntu 12.04 (64 bit). The hardware configuration of the virtual machine is as following :

- Processor : Intel Core i7-3610QM CPU 2.30GHz (we've allowed 8 processors)
- RAM 8030 Mo
- 100 Go of storage

The software configuration of the virtual machine is as following :

- JRE 6.
- Freepastry 2.1

We have used FreePastry’s simulation mode to create a whole community in a single JVM. Additionally, we've based our tests on the past tutorial program that we've found in Freepastry's web site. We've adapted it according to our needs to evaluate how Freepastry react regarding each one of our requirements.

FreePastry's Simulator is a Discrete Event Simulator. It's capable of executing freepastry applications without modification to the source code (Other than the initiation code: Such as the PastryNodeFactory). The code is executed faster than real-time. The simulator uses a virtual clock, not the computer's clock. So if we have the CPU/memory, a simulated environment can run much faster than a real clock. Or if the simulated network is too large or resource intensive to be simulated in real-time, then the accuracy won't be affected by insufficient resources, the virtual clock only advances when there are no immediate tasks to be executed. Contrary to the previous point, the simulator can be executed with the system clock. This may be useful to interactive applications that run on a "human" timescale. Because freepastry supports a "real-time" clock, in the case of a powerful enough computer, it's possible to simulate interactive applications such as games and real-time collaborative applications without using multiple physical machines and a real network.

It can also accept a variety of topologies for network latency. Freepastry ships with 3 topologies: Euclidean (Planar), Spherical, and GenericNetwork topologies. The Euclidean and Spherical topologies use geometric equations to determine latencies between nodes. The GenericNetwork accepts a latency matrix from a file that can be used to construct transit-stub, or more complex topologies.

The simulator simulates delay, and message loss in the case of churn (IE, a node failing with messages in it's queues). The ring grows really slowly if we use the modern consistency leafset/join protocols which are not necessary for the simulator. These protocol guarantee consistency in the face of temporary routing anomalies, which the simulator doesn't simulate.

The proximity is calculated based on the simulator used. In the Euclidean and Sphere ones, it is taken from the geometrical distance. The GT-ITM topology takes it from a file. The unit is mills. In this last case, the way it works is that every time a message is simulated, it looks at the table from A to B, and sets a timer to deliver the message at that time. So if it takes multiple simulated hops, then the time it takes to arrive at the destination is the sum of all of the hops.

The simulator uses one thread for all of the nodes we are simulating. Rather than use a semaphore, it's recommend to schedule the events we want to occur in our simulation as events.

The basic idea behind the simulator is there are a bunch of tasks on a priority-queue sorted by the time they should be executed. To simulate network delay we lookup the delay between the sending/receiving node and schedule a task for that message to be delivered after that amount of time. At each simulation step, we look at the first element on the queue. If the time needs to be advanced to execute that task, we advance the time.

3.4.2. Evaluation Results

3.4.2.1. Storage

3.3.2.4.1. Feature description

The first aspect that we've tested is the ability of storing and retrieving files in Freepastry's DHT [29]. We've started with the storage point because we consider that it represents the core feature of a DSS. Our goal from this tests is to check :

- If Freepastry' DHT is capable of storing and retrieving files of different size,
- The time needed to store and retrieve a file

3.3.2.4.2. Test description

In order to answer the previous questions, we've made some modifications in Freepastry. First of all, we've modified PastContent class which represent an object that can be stored in the DHT. We've added a bite array that will contain the content of a file.

In order to join the overlay ring, a node needs to exchange several messages with nodes contained in the ring to notify theme of its presence and to create its own routing table. Once a node has successfully joined a ring, it will send some messages in order to update its routing table. The test program start with creating a virtual ring containing a fixed number of Freepastry nodes. The number of nodes is defined as a parameter of the program. We create a past application in each node. Once all nodes are created and ready, we pick a random past application on a random node in order to store a file (PastContent object) in the DHT and we keep the identifier of the object (as a key) for the retrieval section of the program. To test simultaneous storage and retrieval in Freepastry, we pick a fixed number of past applications each time to store and to retrieve files. Concerning the retrieval part, we pick a random application and we use past lookup method to search for an already stored key. This way we can test PAST when the node doesn't know the location of the node storing the data.

3.3.2.4.3. Variables description

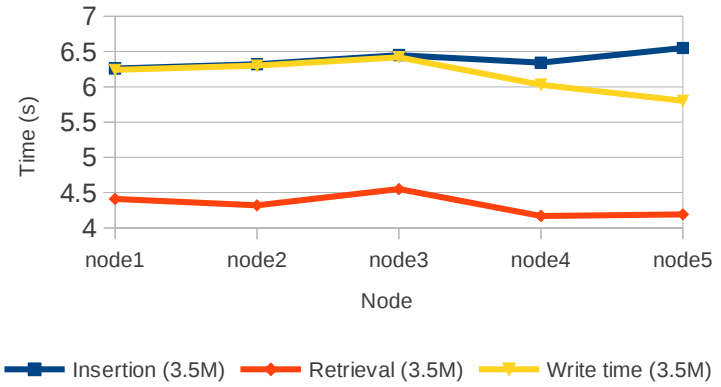
The variables that we took into account for the tests are : file size and number of storage and retrieval at the same time. Here are the parameters that we've set to do the following tests :

- direct_simulator_topology = gt-itm : simulator topology
- pastry_protocol_consistentJoin_max_time_to_be_scheduled : the amount of time it takes the transport layer to find a node faulty (=150000)
- p2p_past_messageTimeout : time out defined for past message. (=30000)
- there is no replication
- there is no cache system
- Number of nodes = 500.

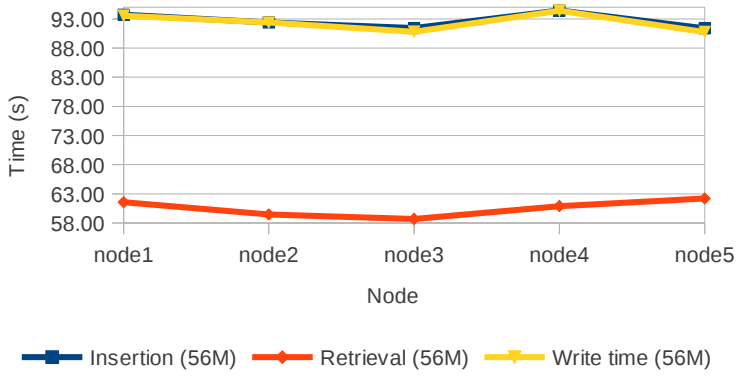
We've used the GT-ITM topology for the simulator and we've used a text file containing the latencies between each pair of nodes. Giving that the simulator is a single threaded program and there is a lock at the time of storage, we've noticed that when we use the same environment for all nodes, we've some delay caused by concurrent access to IO operations. For several times we needed to wait for several nodes to finish their insert operation before getting the response concerning the operation. It seems that all nodes are being executed in the same thread. To resolve this problem and simulate a real system where nodes will be in different machines, we've created for each node a new environment which ensures a new thread for each node.

3.3.2.4.4. Analysis of the results

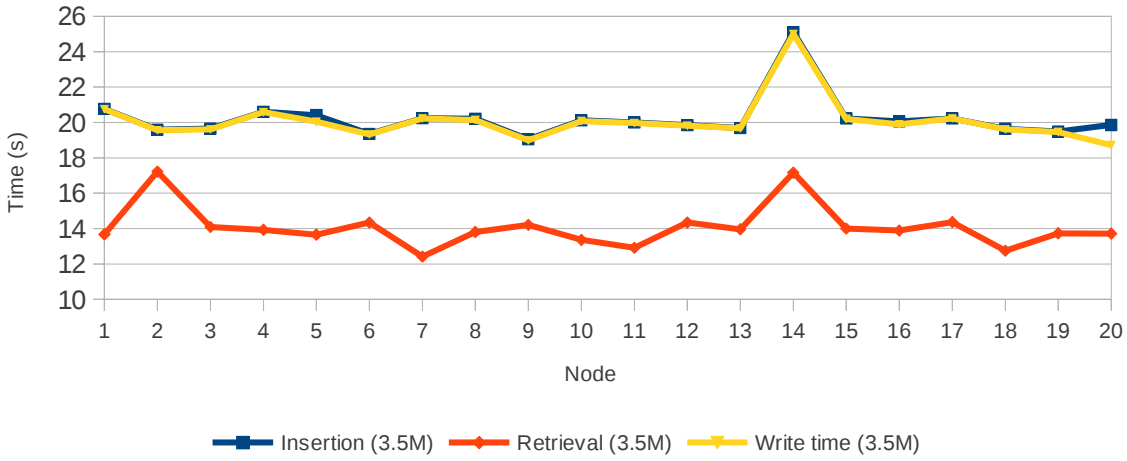
(a) Insertion and retrieval of 5 files (3.5M).
Nodes execute each operation at the same time.



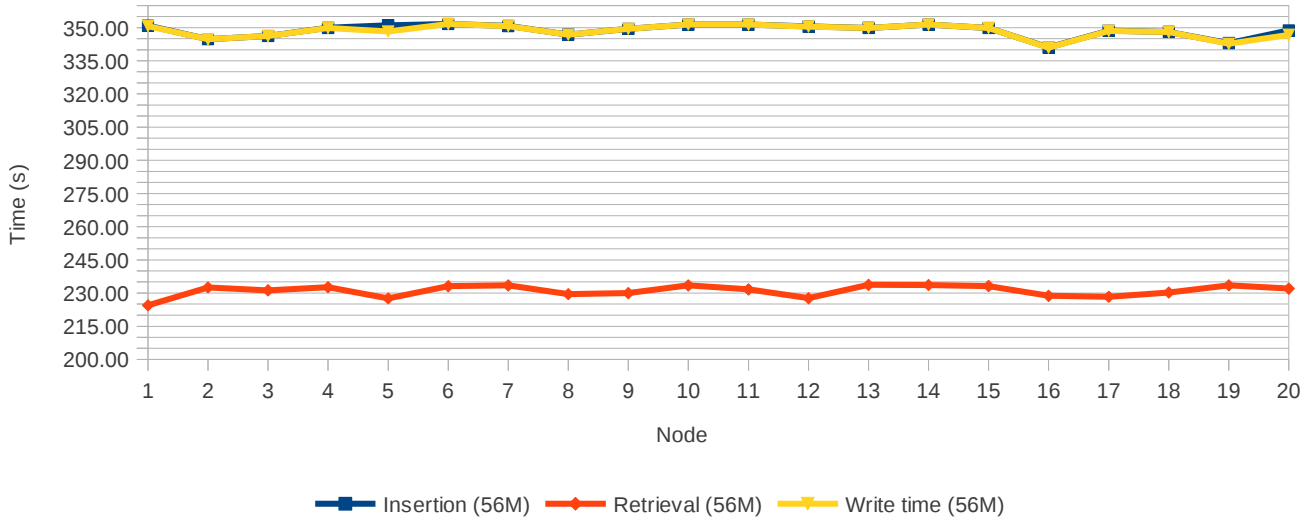
(b) Insertion and retrieval of 5 files (56M).
Nodes execute each operation at the same time.

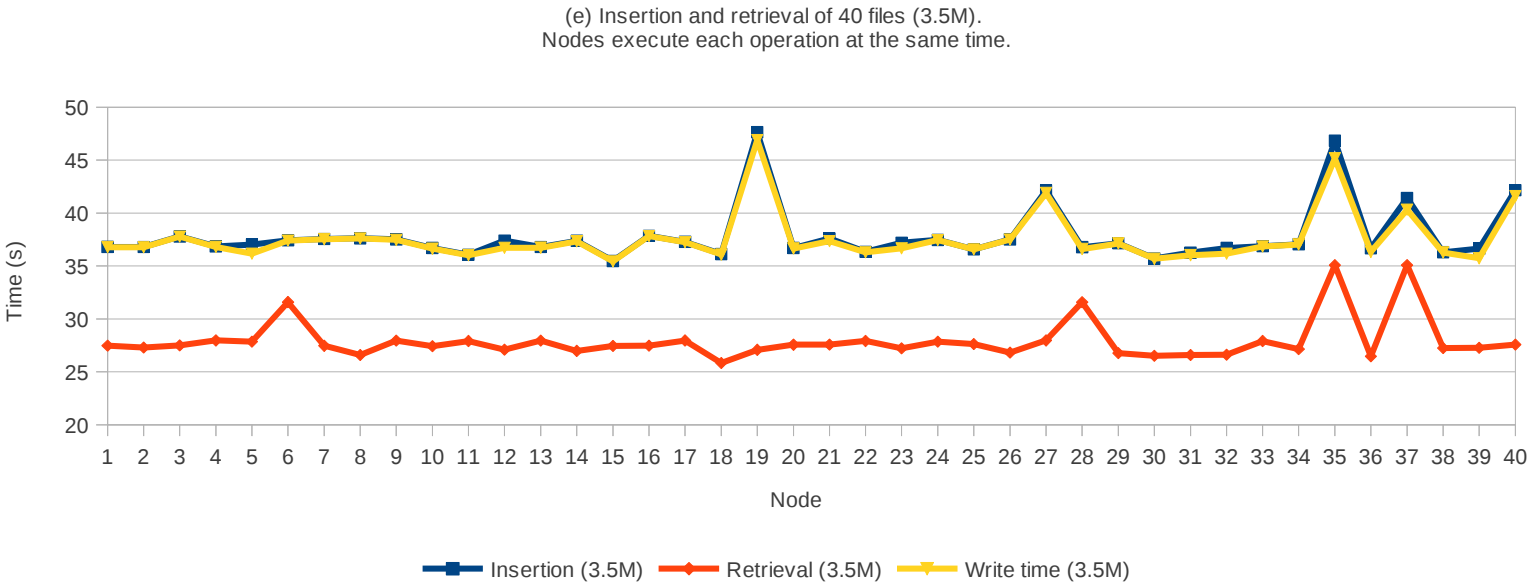


(c) Insertion and retrieval of 20 files (3.5M).
Nodes execute each operation at the same time.



(d) Insertion and retrieval of 20 files (56M).
Nodes execute each operation at the same time.





According to the results above, we can note that the time needed for each node to insert or retrieve data is approximately the same. The cases when we have differences of the order of 5s for example, can be explained by the concurrent access of IO operations. We can notice also the effect of this later when we increase the number of files. When we try to insert 20 or 40 files at the same time, there are more concurrent access to IO operations and it's reflected in the time needed for each data to be stored or retrieved. Another point that we can get from the results is that there is no much difference between the time needed to write an object to disk by the node storing the object “write time” and the time needed to call insert method by a random node and get the acknowledgement for the insert operation from the node storing the object.

We need to note that in this tests, the nodes doesn't know the location of the data and there is some traffic required to locate the data before retrieving it. We need to note also that in this tests, we've used a thread per node to prevent the locking operation performed at the time of writing the data. If we had used a single thread, results will be different than the ones above because nodes will block each one another and we'll get the acknowledge response once several nodes had finished their insertion.

There is two ways to get an object from the node storing it. The first one is by calling the lookup method which will locate the node storing the object and will retrieve the data from it (this is the method that we've used for our tests). The second method is to use the lookupHandles to get the handles responsible for an object and use one of the handles to fetch the object from the node storing it. In order to select an appropriate handle, we can use the isAlive method to check if the handle is alive and after that use the proximity method to select the nearest node from the node requiring the object.

For the tests, we've used lookup method which doesn't chose the best downloading location to get an object. This approach is suited for small files but in the case of middle and big files it's better to use the second method because we'll be getting objects from the nearest node. For the test we considered that there is no churn, our main goal was to prove that is possible to store and retrieve files in freepastry's DHT. However, for a real environment, it's better to use the second method because it will allow us to check the availability of the data before trying to get it. Performance won't be affected because with the first way we search for the location of the object before downloading it and with the second way we search for the location of handles responsible for the object before downloading it directly from the node storing it.

3.4.2.2. Scalability

3.3.2.4.1. Feature description

Another important aspect that Freepastry's DHT [29] needs to handle correctly is scalability. We need to compare the results from the previous tests that are based on a limited number of nodes in the community and the ones where we've a big number of nodes. To say that scalability is handled correctly, the time needed for storage or retrieval must not be affected by the number of nodes in the ring.

3.3.2.4.2. Test description

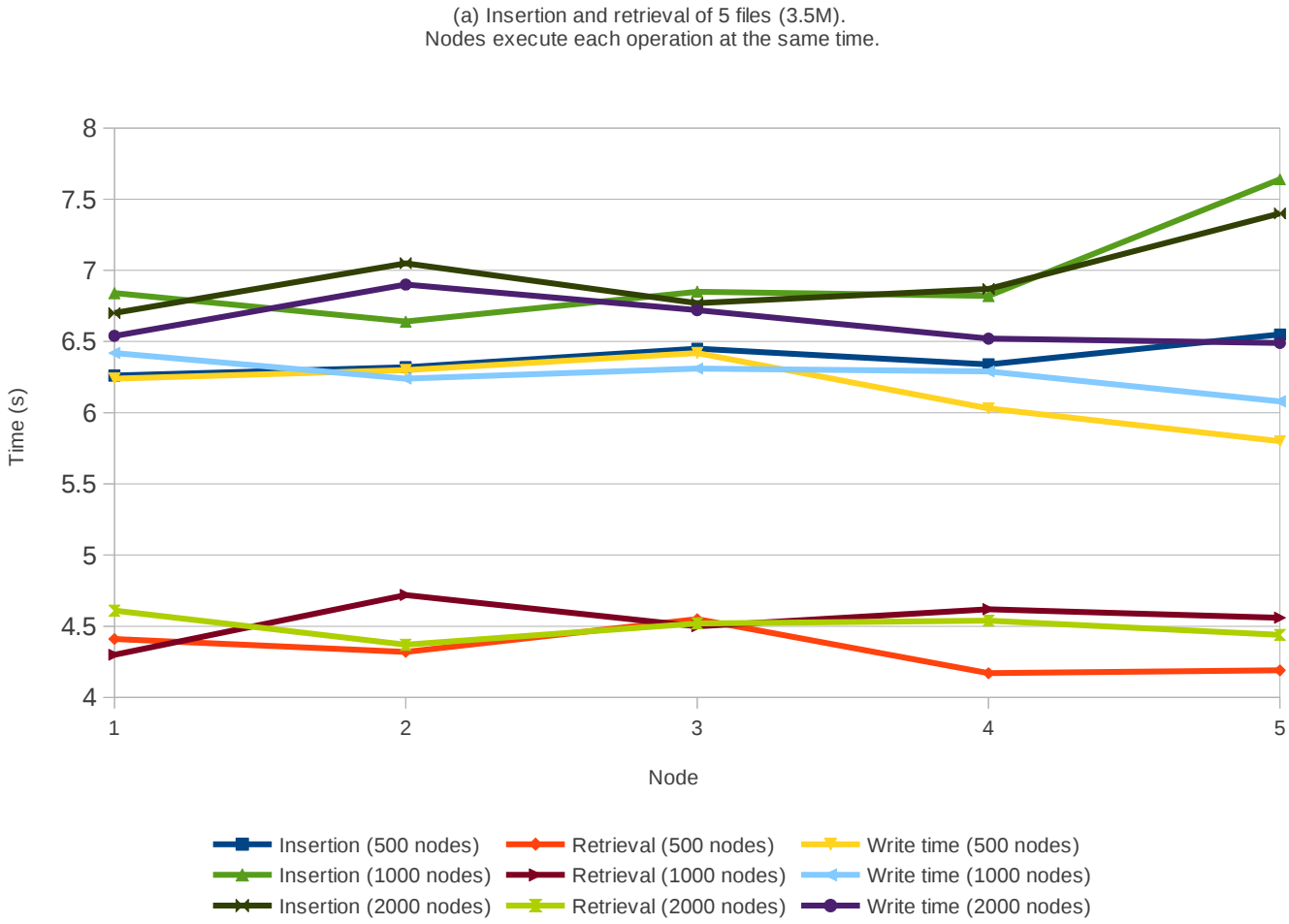
To test scalability, we'll use the same program as for the storage tests, the only difference is the number of nodes. Each time, we'll increase the number of nodes an see how the time needed for insertion or retrieval is affected.

3.3.2.4.3. Variables description

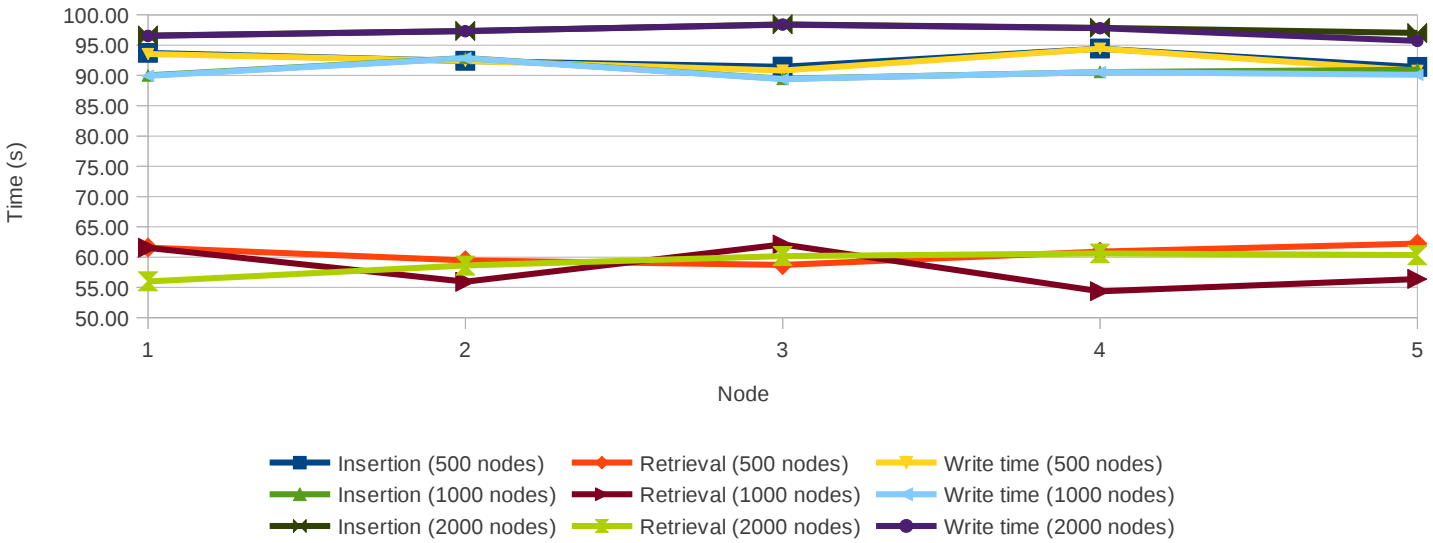
The variable that we took into account for the tests is : the size of the community. We've tested 500, 1000 and 2000 nodes and checked how the system reacts.
Here are the parameters that we've set to do the following tests :

- direct_simulator_topology = gt-itm : simulator topology
- pastry_protocol_consistentJoin_max_time_to_be_scheduled : the amount of time it takes the transport layer to find a node faulty (=150000)
- p2p_past_messageTimeout : time out defined for past message. (=30000)
- there is no replication
- there is no cache system
- Number of nodes = 500, 1000 and 2000.

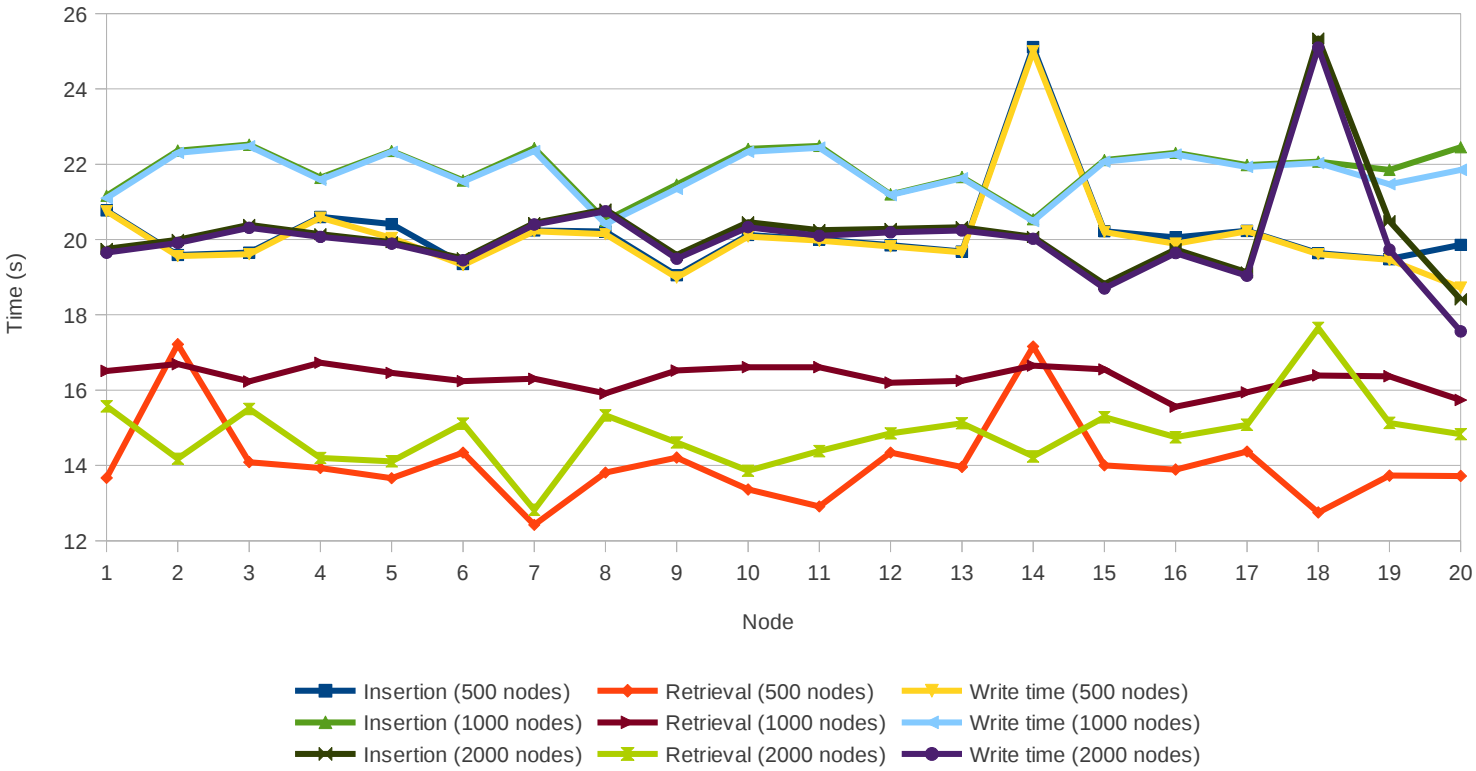
3.3.2.4.4. Analysis of the results



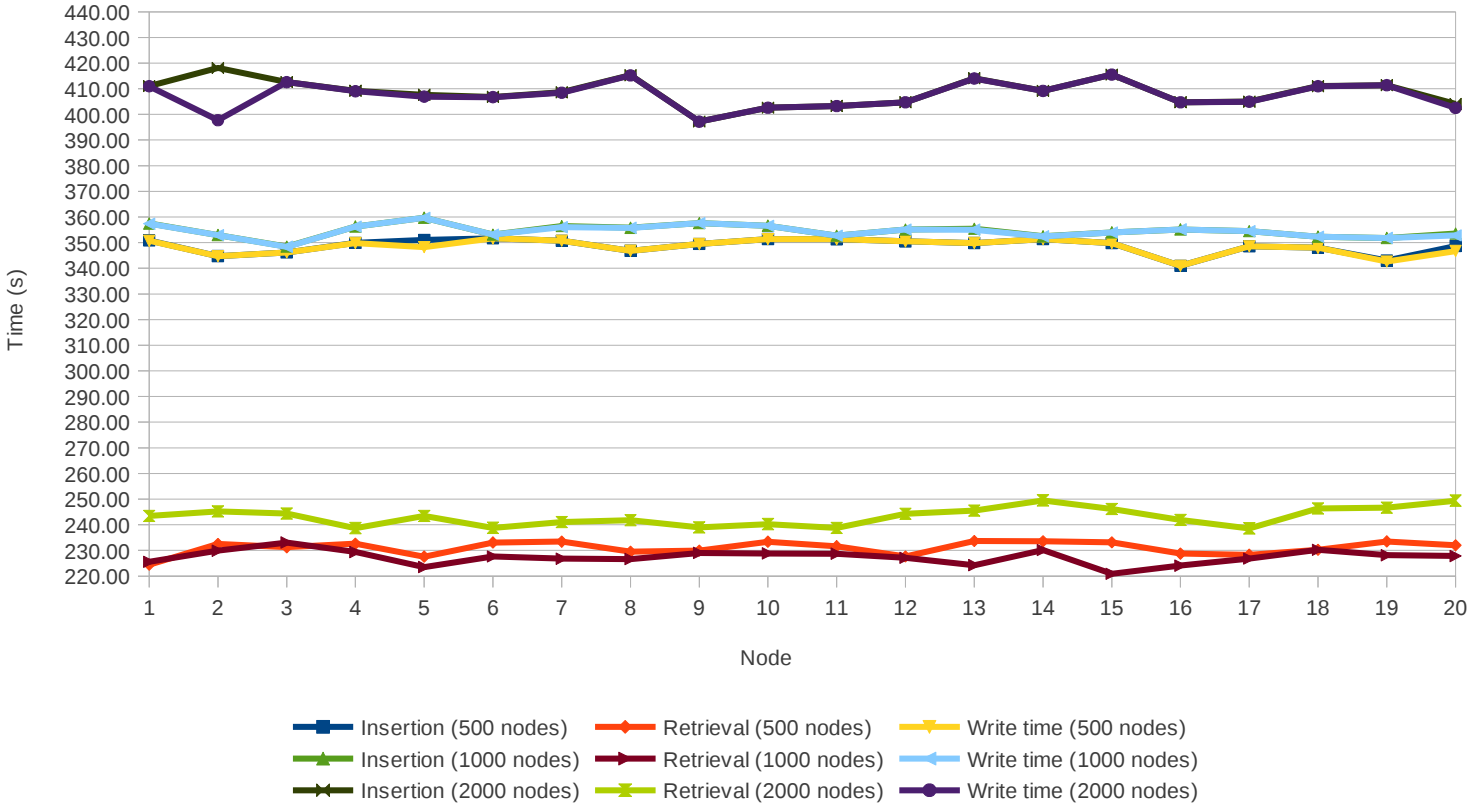
(b) Insertion and retrieval of 5 files (56M).
Nodes execute each operation at the same time.



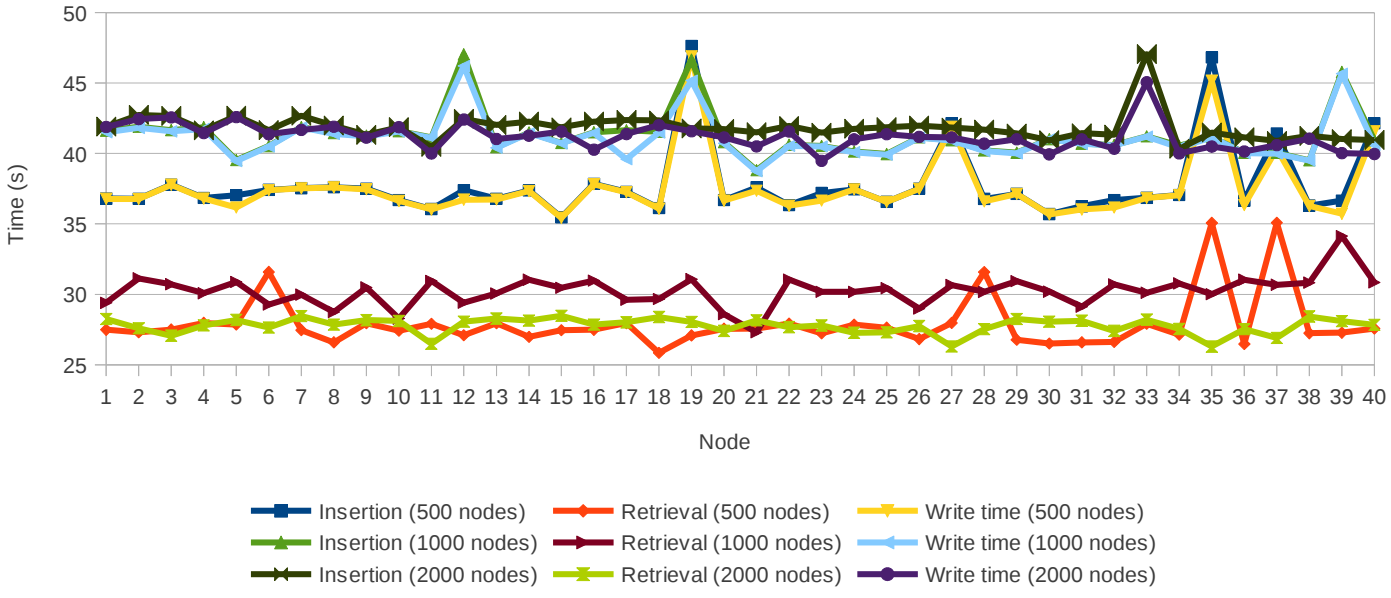
(c) Insertion and retrieval of 20 files (3.5M).
Nodes execute each operation at the same time.



(d) Insertion and retrieval of 20 files (56M).
Nodes execute each operation at the same time.



(e) Insertion and retrieval of 40 files (3.5M).
Nodes execute each operation at the same time.



Increasing the number of nodes in the ring have two direct impacts in the freepastry overlay network. The first one is the increase number of messages that will be exchanged between nodes to stabilize the network and the second one is the increase number of hopes needed to get or put an object. In a system that handles correctly scalability, these two aspects must be transparent regarding insertion and lookup operations. Indeed, the time needed for each operation will increase with the number of nodes but difference must not be considerable in order to ensure scalability.

According to the results above, we can note that in the case of a small file (3.5 M), scalability doesn't have an important impact on the time needed for insertion or retrieval. The difference can be explained by the fact that there is more nodes and the number of hopes needed to get to the destination node is bigger. In the charts above we can see that the difference is insignificant.

In the case of big files (56 M) and 5 operations at the same time, we can note the same results. However in the case of 2000 nodes and 20 operations, we can see that the difference between the insert operations is of the order of 50s. This can be explained by the following facts :

- When we store an object using freepastry's writing method, we generate bigger files than the originals. For example, for a file of 56M, we'll store a file of 111M.
- For the tests, we've used a single machine and to store 20 files of 56M, we write to disk 20 files of 111M at the same time. These concurrent writing operations will have an impact on the time needed to write a single file.

In a real environment, the writing operations will be executed in several nodes and we won't have this delays related to concurrent access to IO operations.

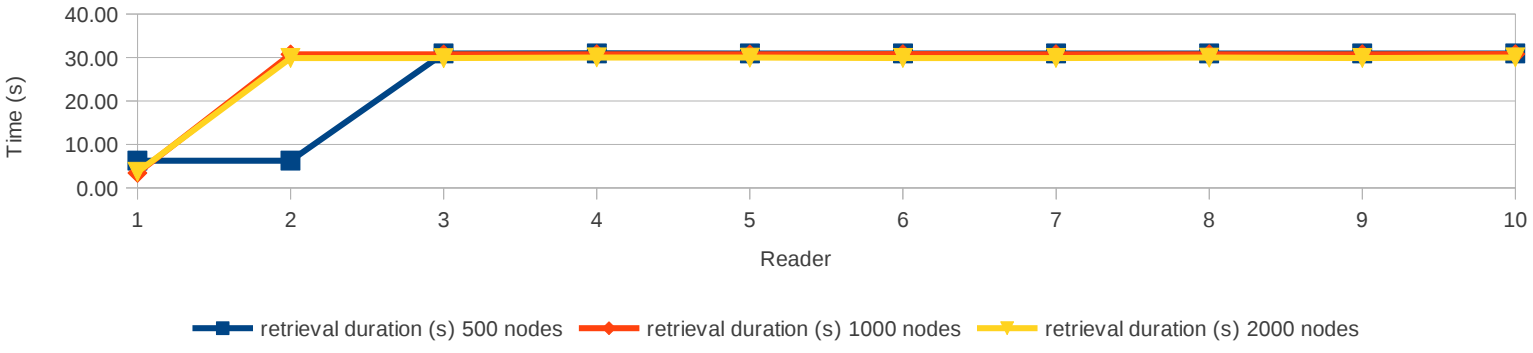
We need to note that it wasn't possible to test the case of 40 files (56 M), because it requiters a lot of resources and the test machine wasn't able to execute the test.

3.4.2.3. Scalability and Multiple readers

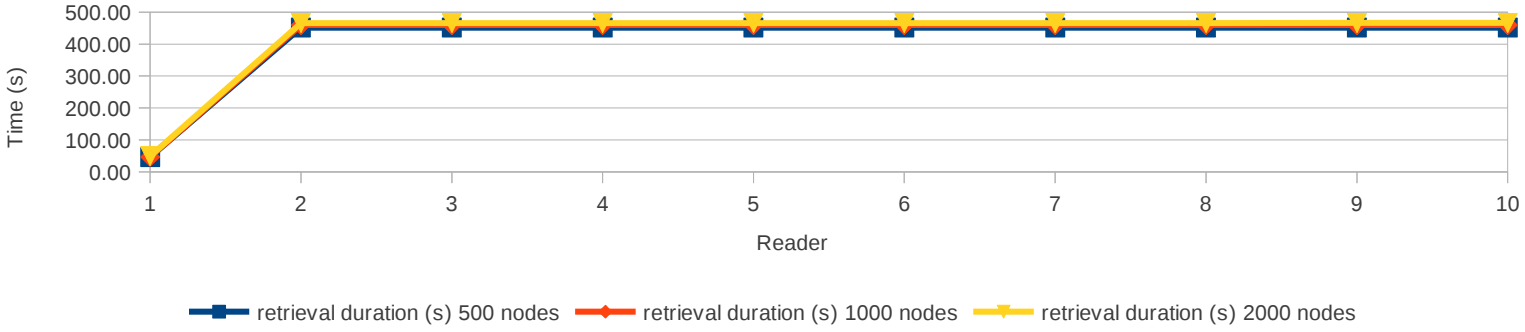
Another aspect that we've tested is scalability in the case of multiple readers. This time we inserted 2 files in the DHT and tried to read these files several times at the same time. We've tested 10 and 20 reads at the same time and for different size of the ring : 500, 1000 and 2000.

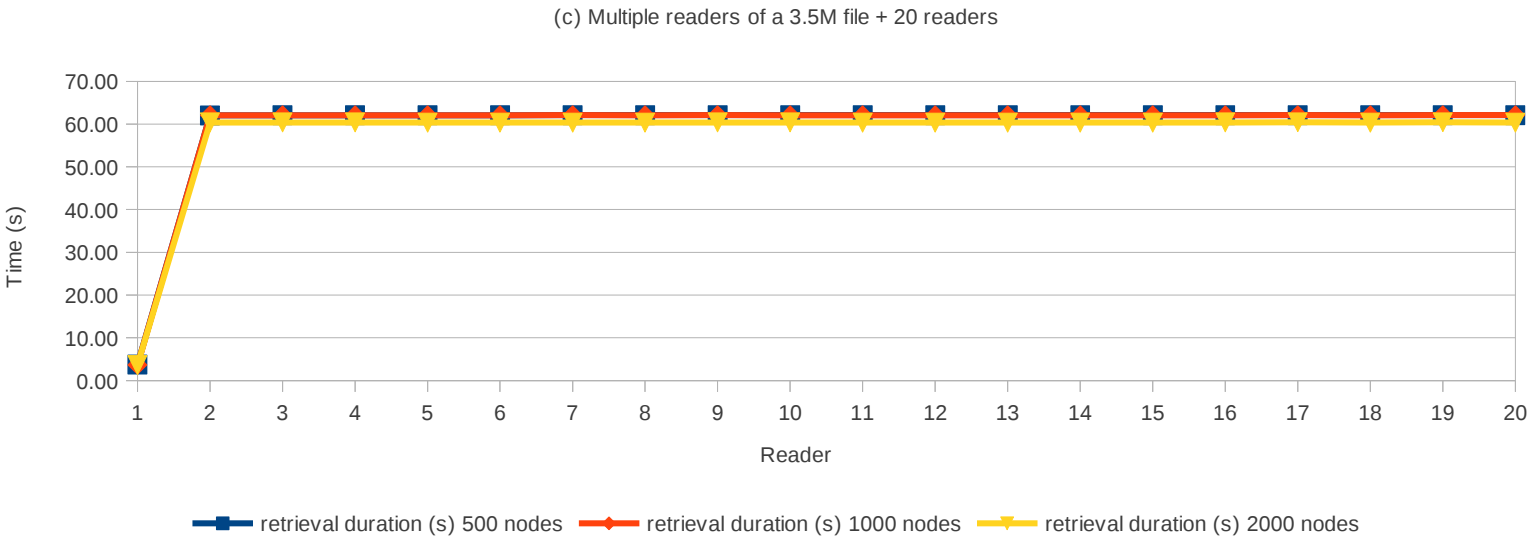
3.3.2.4.1. Analysis of the results

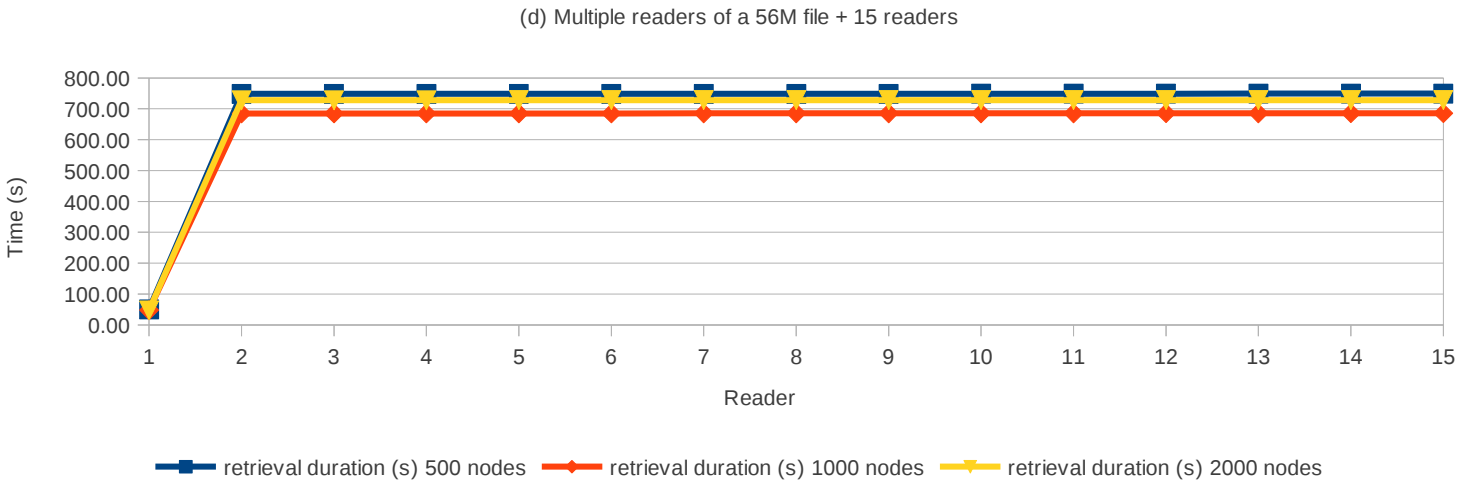
(a) Multiple readers of a 3.5M file + 10 readers



(b) Multiple readers of a 56M file + 10 readers







Thanks to the results above, we can see that scalability doesn't affect multiple readers. Indeed, there is a difference related to the number of nodes but it can be explained by the fact that the lookup method need more time to reach the node storing the data. However, we need to note that giving that all nodes read from the same node and the method that retrieve the file use a blocking IO mechanism to respond to a single request at a time which explain the time needed to answer the first node and the following.

This issue can be improved by modifying the get method and using multiple threads to attend a bigger number of requests.

3.4.2.4. Replication & Churn

3.3.2.4.1. Feature description

Replication is the mechanism that will ensure availability of data in case of nodes failure. Each time a data will be stored in the distributed storage system, it will be replicated according to a fixed number of replicas. Churn represents the factor of unavailability of nodes. These later can leave the ring whenever they want and the system must react in consequence to be able of handling the already stored data. Nodes send periodically messages to check if a node is still alive. When a node doesn't acknowledge of its presence, nodes exchange messages to update their routing tables and to locate a suitable replica for the data.

3.3.2.4.2. Test description

For this tests, we'll simulate churn to see how the system react in case of nodes failure and we'll check if replication works

correctly in freepastry. To test the performances of insertion and retrieval in the case of replication and churn, we've used the same part of the program (as in the previous tests) that is responsible of the creation of a ring, initializing each node, creating a past application for each node and inserting a number of files. The remaining part of the program will be in charge of destructing some nodes containing the stored files. Once these nodes have leaved the ring, we pick some random past applications and we lookup for the handles of an already stored files. When the lookupHandles method get an array of the handles responsible for the file, we check for each handle if it's alive. finally, we use the fetch method with one of the alive handle to get the file. In the list of available handles, we can use the proximity method to select the nearest node to minimize the time needed for retrieval.

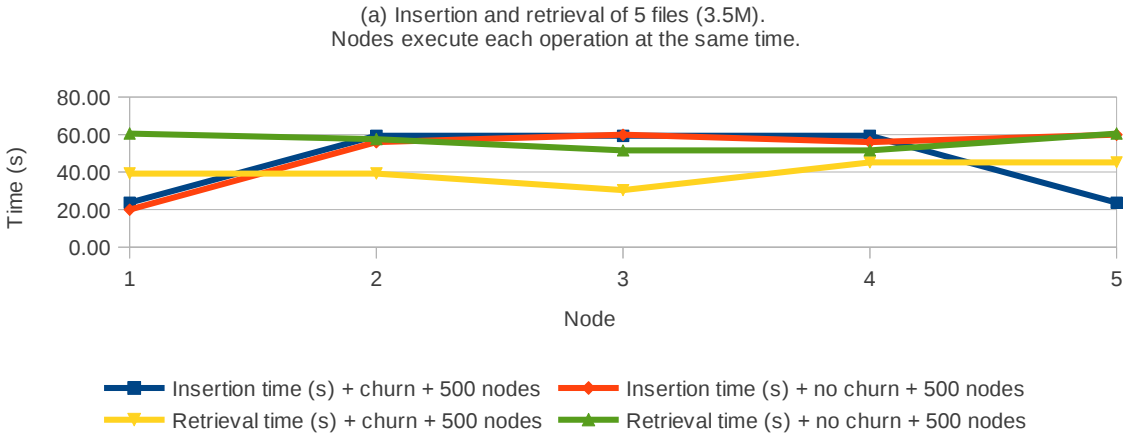
3.3.2.4.3. Variables description

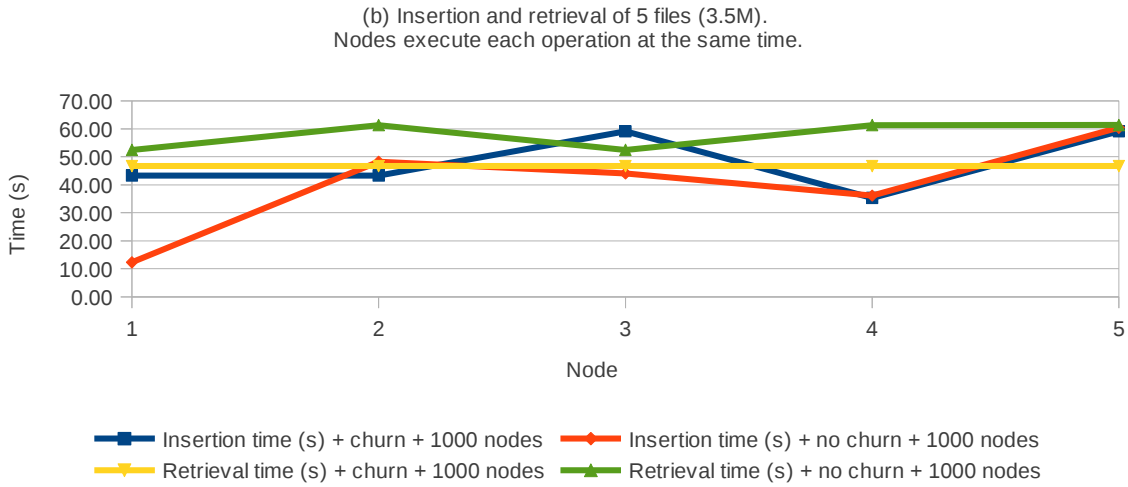
The variables that we took into account for the tests are : the size of the community and the file size.

Here are the parameters that we've set to do the following tests :

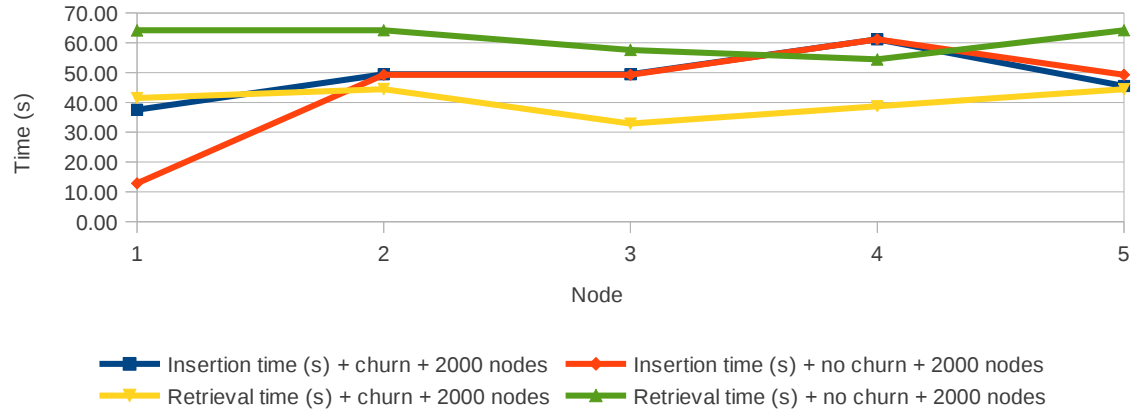
- direct_simulator_topology = gt-itm : simulator topology
- pastry_protocol_consistentJoin_max_time_to_be_scheduled : the amount of time it takes the transport layer to find a node faulty (=150000)
- p2p_past_messageTimeout : time out defined for past message. (=30000)
- replication = 2
- there is no cache system
- Number of nodes = 500, 1000 and 2000.

3.3.2.4.4. Analysis of the results

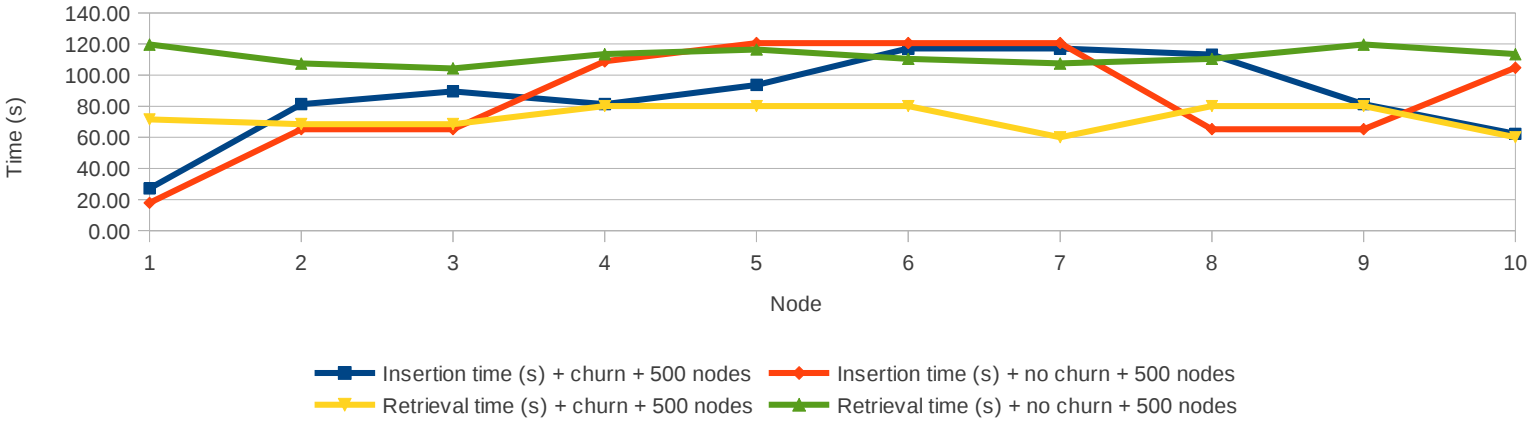




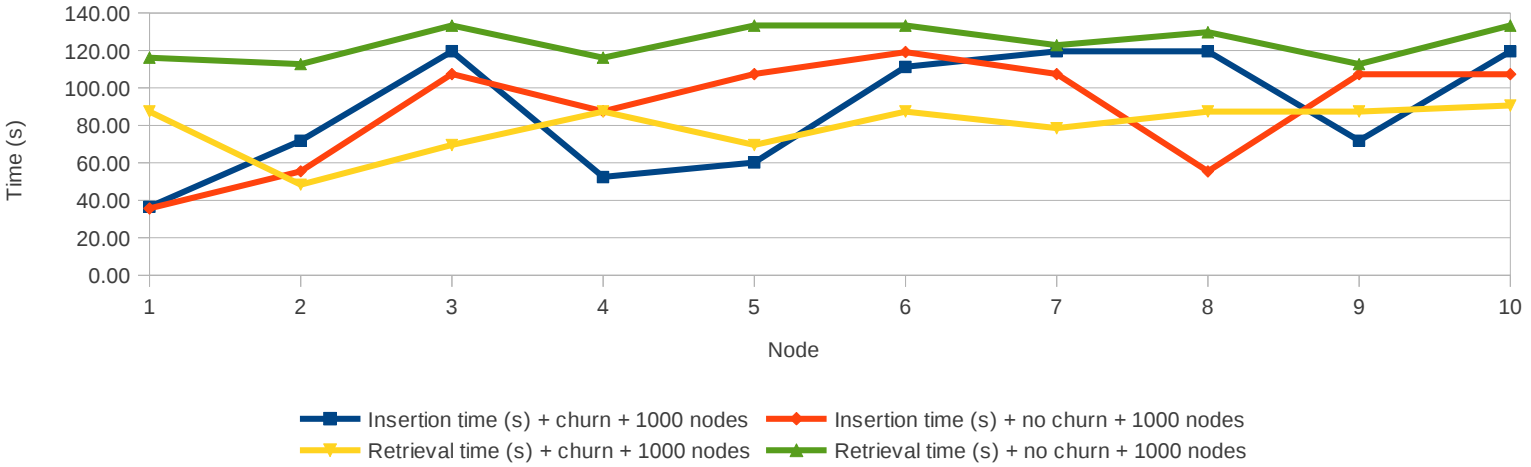
(c) Insertion and retrieval of 5 files (3.5M).
Nodes execute each operation at the same time.

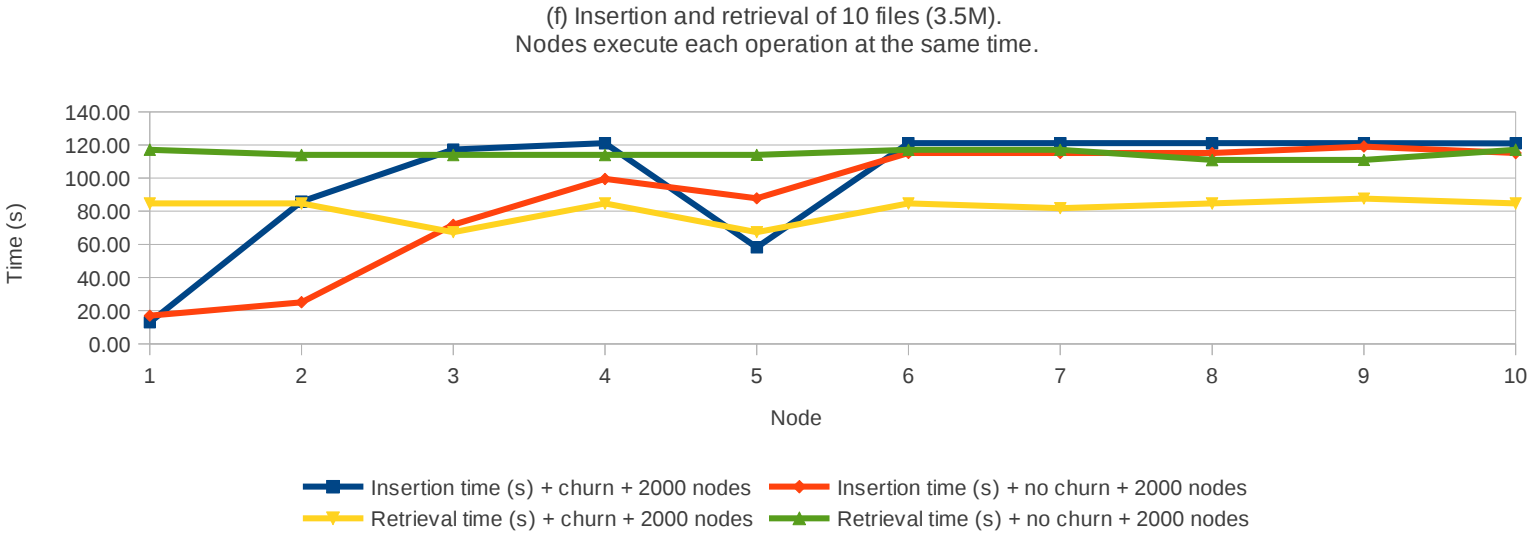


(d) Insertion and retrieval of 10 files (3.5M).
Nodes execute each operation at the same time.



(e) Insertion and retrieval of 10 files (3.5M).
Nodes execute each operation at the same time.





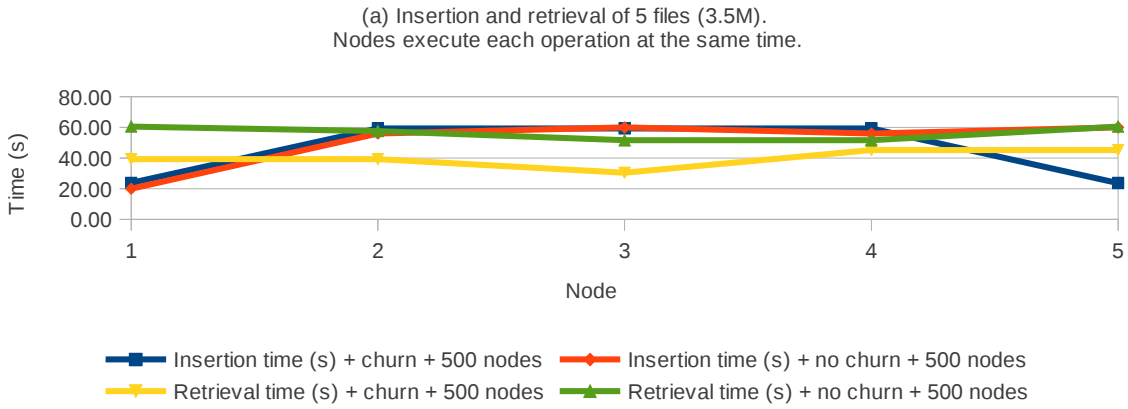
There is an important point that we need to explain concerning these results. The difference between the figures concerning the time needed for insertion and retrieval in these tests and the previous ones, can be explained by the fact that in order to test churn in the simulator, we must use one environment for all nodes, thus a unique thread for all nodes. To test a node failure or departure, we've used the method environment.destroy(). When we clone the environment for all nodes, the destroy method seems to be taken into account by the other nodes and they exchange messages to stabilize the network but when we use a new environment for each node, the destroy method is ignored. Giving that there is a lock method used for IO operations, sometimes nodes doesn't send a response directly after the insertion operation but they are blocked until other nodes have finished their insertions. For example, if the time needed for an insertion is 90s for each node, sometimes we'll get the acknowledgement of the insertion operation after 180s for the three nodes at the same time. If there is other nodes that try to insert files at the same time as the previous nodes, they will wait for the three previous nodes to finish before trying to insert their files at the same time. This is why we can see sometimes 1100s for example for an insertion, in fact the real time needed in a real environment is 100s but giving the concurrent access to IO operation a node will wait for the other 10 nodes to finish their jobs before finishing his operation.

According to the results above, if we take into account the remark from the previous paragraph, we can note that churn doesn't affect the availability of files, it's possible to continue retrieving data even in the presence of failure or departure of nodes. However, we can notice that the time of insertion or retrieval in the case of no churn is sometimes bigger than the time in the case of churn. This can be explained by the fact that in the presence of churn there will be some replicas missing before recreating another ones thus the lookuphandles method will return the available handles quicker. We've done the same tests for files of 56M and we've noticed the same behavior.

3.4.2.5. Versionning and Atomic modification

As we've seen in the previous tests, the writing method uses a blocking mechanism to treat one request at a time. This way it can ensure that modification will be atomic and concurrent writes won't be the source of conflicts.

In the chart below we can see that the time needed to insert an object in the case of churn for the first and the last node is 20s but in the case of the other three nodes is 60s. In reality, the time needed to insert an object is 20s, but giving that there is a blocking mechanism, nodes 2, 3 and 4 have not been able to send the acknowledgement for the insertion until the three have finished (20s x 3 = 60s). Therefore, we can conclude that a blocking mechanism is used in the writing method and it ensures atomic modification.



Another requirement that needs to be respected by freepastry's DHT in order to say that it could be used for CoDeS is versionning. We need to be able to insert different versions of a file. Natively, PAST implementation in freepastry doesn't handle multiple versions of a file. Thus, we've modified the code to be able to do it.
Freepastr's PAST uses the key of an object as its physical name. In order to be able to manage multiple versions, we've added to the physical name of the object a separator '°' and the time stamp of the insertion. The name of the file will be : key°timestamp instead of key. This way we can have multiple versions of a file. But this modification is not sufficient, because when the storage implementation will search the disk for a file named as the key it won't find it. Thus, we've modified the storage implementation to lookup for files which name start with the key and we select the one with the bigger time stamp. The time needed to lookup for the files in the disk is insignificant regarding the time needed to get an object, consequently it won't affect performances. This method allow us to have multiple versions but we can get only the last version. More modifications needs to be done in order to get a specific version.

3.4.2.6. Garbage collector

The garbage collector offered by freepastry [29] can be used to manage the deletion of objects in the distributed storage system using timeout. This package will allow us to handle storage space efficiently. Individual objects are each given timeout times. These are used to determine when space can be reclaimed. Applications must periodically invoke the refresh() operation on all objects which they are interested in. Otherwise, objects may be reclaimed and will be no longer available.

3.4.2.7. Security

Freepastry API offers an interface named PastPolicy which represents a policy for Past. It's called whenever the local node is told to replicate or validate an item. This allows for applications to control replication and object validate behavior, permitting behavior specific for mutable or self-authenticating data.

4. Conclusion

The goal of this research was to provide a distributed storage system for CoDeS [22] a contributory computing model, where users form a community by contributing their resources to be used collectively in a decentralized and self-managed way. For this purpose, we've studied the existing distributed storage systems and we've analyzed each element constituting them and each feature characterizing them. During the analysis we've took into account the functional and non functional requirements for our research to be able to select the best solution.

We've considered that PAST [8] is the best distributed storage system that could be integrated in CoDeS because it shares a lot of common aspects with a contributory computing model. Additionally, it implementation is available in freepastry [29] which is the system used in CoDeS prototype. However, further tests needed to be accomplish in order to prove the validity of this solution.

According to the tests and evaluations that we've done on freepastry's implementation of PAST, we can conclude that it can handle correctly the following features : scalability, availability, fault-tolerance and consistency. This solution is capable also of handling the functional requirements : storage and retrieval of different files size and atomic modifications. However, to be integrated with CoDeS, some improvements need to be done as managing different file versions and security by implementing an authentication mechanism. Additionally, we've seen that the solution is not capable of handling multiple reads at the same time which is an important aspect for distributed storage system giving that in the case of service deployment multiple nodes would request the same data at the same time. Finally, we've conducted the tests on a single server and we think that it could be interesting to compare this results with tests executed on a real test environment as Planetlab.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. SIGOPS Operating Systems Review 36, SI, 1-14.

[2] BRAAM, P. J. 2002. The lustre storage architecture. Cluster File Systems Inc. Architecture, design, and manual for Lustre. http://www.lustre.org/docs/lustre.pdf.

[3] CARNS, P. H., LIGON III, W. B., ROSS, R. B., AND THAKUR, R. 2000. PVFS: A parallel file system for Linux clusters. In Proceedings of the 4th Annual Linux Showcase and Conference. USENIX Association, Atlanta, GA, 317–327.

[4] CASTRO, M., ADYA, A., LISKOV, B., AND MYERS, A. C. 1997. HAC: Hybrid adaptive caching for distributed storage systems. In ACM Symposium on Operating Systems Principles (SOSP). Saint Malo, France, 102–115.

[5] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. 2001. Freenet: A distributed anonymous information storage and retrieval system. Lecture Notes in Computer Science 2009, 46+.

[6] DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND WELCH, B. B. 1994. The bayou architecture: Support for data sharing among mobile users. In Proceedings IEEE Workshop on Mobile Computing Systems & Applications. Santa Cruz, California, 2–7.

[7] DINGLEDINE, R., FREEDMAN, M. J., AND MOLNAR, D. 2000. The free haven project: Distributed anonymous storage service. In Workshop on Design Issues in Anonymity and Unobservability. Number 2009 in LNCS. 67–95.

[8] DRUSCHEL, P. AND ROWSTRON, A. 2001. PAST: A large-scale, persistent peer-to-peer storage utility. In HotOS VIII. Schloss Elmau, Germany, 75–80.

[9] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. Oceanstore: An architecture for global-scale persistent storage. In Proceedings of ACM ASPLOS. ACM.

[10] LUA, K., CROWCROFT, J., PIAS, M., SHARMA, R., AND LIM, S. 2005. A survey and comparison of peer-to-peer overlay network schemes. Communications Surveys & Tutorials, IEEE, 72–93.

[11] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. 2002. Ivy: A read/write peer-to-peer file system. In Proceedings of 5th Symposium on Operating Systems Design and Implementation. USENIX.

[12] PLACEK, M. AND BUYYA, R. 2006. Storage exchange: A global trading platform for storage services. In 12th International European Parallel Computing Conference (EuroPar). LNCS. Springer-Verlag, Berlin, Germany, Dresden, Germany.

[13] ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware). Lecture Notes in Computer Science 2218, 329–350.

[14] SATYANARAYANAN,M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers 39, 4, 447–459.

[15] SCHMUCK, F. AND HASKIN, R. 2002. GPFS: A shared-disk file system for large computing clusters. In Proc. of the First Conference on File and Storage Technologies (FAST). 231–244.

[16] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. 2001. Chord: A scalable Peer-To-Peer lookup service for Internet applications. In Proceedings of the 2001 ACM SIGCOMM Conference. 149–160.

[17] THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: A scalable distributed file system. In Symposium on Operating Systems Principles. 224–237.

[18] VAZHKUDAI, S. S.,MA, X., FREEH, V. W., TAMMINEEDI, J. W. S. N., , AND SCOTT., S. L. 2005. Freeloader: Scavenging desktop storage resources for scientific data. In IEEE/ACM Supercomputing 2005 (SC|05). IEEE Computer Society, Seattle, WA.

[19] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. 2003. Tapestry: A global-scale overlay for rapid service deployment. IEEE Journal on Selected Areas in Communications. Special Issue on Service Overlay Networks, to appear.

[20] Martin PLACEK AND Rajkumar BUYYA. A Taxonomy of Distributed Storage Systems. The University of Melbourne.
[21] Srikumar Venugopal, Rajkumar Buyya and Ramamohanarao Kotagiri. A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing . Grid Computing and Distributed Systems Laboratory, Department of Computer

Science and Software Engineering, The University of Melbourne, Australia .

[22] Daniel Lazaro Iglesias. CoDeS: A Middleware for Service Deployment in Contributory Computing Systems. DPCS Research Group , Internet Interdisciplinary Institute and Universitat Oberta de Catalunya.

[23] Jean-Michel Busca , Fabio Picconi , and Pierre Sens. Pastis: a Highly-Scalable Multi-User Peer-to-Peer File System. INRIA Rocquencourt Le Chesnay, France and LIP6, Universite Paris 6 – CNRS Paris, France.

[24] Napster, http://www.napster.com.

[25] Gnutella, http://gnutella.wego.com.

[26] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. Theory of Computing Systems, 32:241–280, 1999.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content addressable network,” in Processings of the ACM SIGCOMM, 2001, pp. 161–172.

[28] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, “Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination,” in Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video, 2001, pp. 11–20.

[29] freepastry, http://www.freepastry.org/FreePastry/.