Títol: Currency Management System:
A Distributed Banking Service for the Grid

Volum: 1/1
Alumne: Xavier León Gutiérrez

Director/Ponent: Leandro Navarro Moldes
Departament: Arquitectura de Computadors
Data: 5 de Julio, 2007

DADES DEL PROJECTE

Títol del Projecte: Currency Management System: A Distributed Banking Service for the Grid

Nom de l'estudiant: Xavier León Gutiérrez
Titulació: Enginyeria en Informàtica
Crèdits: 37.5
Director/Ponent: Leandro Navarro Moldes
Departament: Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Felix Freitag

Vocal: Omer Giménez Llach

Secretari: Leandro Navarro Moldes

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Currency Management System: a Distributed Banking Service for the Grid

Xavier León Gutiérrez

Projecte Final de Carrera (PFC)

July 5, 2007

Facultat d’Informàtica de Barcelona
Univertitat Politècnica de Catalunya

Agradecimientos

En primer lugar me gustaría dar las gracias a mi familia, en especial a mis padres, por haberme apoyado a lo largo de mi vida y darme siempre todo lo necesario para parecerme a una persona de provecho sin pedir nada a cambio. Sin vuestra ayuda no habría llegado hasta aquí.

Muchas gracias a mi director de proyecto, Leandro Navarro, no sólo por guiarme con sus consejos e ideas a lo largo de estos meses, sino por tener siempre palabras de soporte y ánimo en los momentos necesarios.

Quisiera dar las gracias especialmente a Gemma, la persona que me ha acompañado a lo largo de los últimos años, por la paciencia y comprensión que ha tenido así como los momentos compartidos y el respaldo que me ha mostrado incondicionalmente.

Por último, agradecer a los amigos y compañeros que he conocido a lo largo de mis estudios por hacer mi estancia en esta facultad mucho más llevadera. Los momentos que me habéis ofrecido son, realmente, inolvidables.

Contents

List of Figures xi

List of Algorithms xiii

List of Tables xv

Preface 1

Introduction 1
Context 2
Goals and Scope 2
Project Overview 3
Document Organization 4

I Concepts and Survey of Related Work 5

1 Virtual Currency management and payment protocols 7

1.1 Introduction 7
1.2 Currency Market 8
1.2.1 Closed Currency Market 8
1.2.2 Open Currency Market 10
1.3 Multiplicity of currencies 11
1.4 Currency representation 11
1.4.1 Account balance based 12
1.4.2 Token based 12
1.5 Currency storage 13
1.5.1 Local 13
1.5.2 Remote 14

1.6 Transaction Protocol . . . 15
1.6.1 Bank intervention . . . 15
1.6.2 Ordering in transactions . . . 16
1.6.3 Transaction amount . . . 18
1.7 Security Considerations . . . 18
1.7.1 Authenticity . . . 18
1.7.2 Privacy . . . 19
1.7.3 Counterfeiting . . . 20
1.8 Survey and Comparison . . . 20
1.8.1 PeerMint . . . 21
1.8.2 Karma . . . 22
1.8.3 PPay and CPay . . . 23
1.8.4 GridBank and Tycoon Bank . . . 24
1.8.5 Hash Chains . . . 24

2 Distributed Hash Tables 27
2.1 Introduction . . . 27
2.2 DHTs structure . . . 30
2.2.1 Overlay network topoly . . . 30
2.2.2 Routing information . . . 31
2.3 Replication schemes . . . 32
2.3.1 Replication as a technique to achieve fault-tolerance . . . 32
2.3.2 Replication mechanisms in DHTs . . . 33
2.4 Survey and comparison . . . 37
2.4.1 Chord . . . 37
2.4.2 Pastry and Tapestry . . . 38
2.4.3 Distributed k-ary System . . . 40

3 Consistency in DHTs 43
3.1 Introduction . . . 43
3.2 Consistency mechanisms . . . 44
3.2.1 Key Consistency . . . 44
3.2.2 Data Consistency . . . 45
3.3 Survey and comparison . . . 45
3.3.1 Atomic Ring Maintenance . . . 45

3.3.2 Token based root authorization 46
3.3.3 Etna: consensus over DHT 47
3.3.4 Atlas P2P Architecture (APPA) 48

4 Mutual Exclusion in DHTs **51**
4.1 Introduction 51
4.2 Mutual Exclusion properties 51
4.3 Classification of Distributed Mutual Exclusion algorithms 52
4.4 Survey and comparison 53
4.4.1 Sigma algorithm 53
4.4.2 End-to-End and Non End-to-End protocols 54

II The Currency Management System **57**

5 General Overview **59**

6 System requirements analysis and specification **61**
6.1 System requirements 61
6.1.1 User requirements 62
6.1.2 System requirements 63
6.1.3 Infrastructure requirements 65
6.2 System specification 66
6.2.1 Actors 66
6.2.2 API Specification 66
6.3 System Model 73

7 System Architecture **75**
7.1 Grid4All Market Place Architecture 75
7.2 Payment Module logical view 77
7.3 Currency Management System Architecture 78
7.3.1 Deployment View 78
7.3.2 Component View 79
7.3.3 Roles 81

8 Design and Implementation **83**

8.1 Key Based Routing Layer 83
8.1.1 Key Based Routing API 85
8.2 Mutable Consistent Layer 86
8.2.1 Mutable Consistent DHT API 87
8.2.2 Mutable Identifier Space 89
8.2.3 Consistency Mechanism 90
8.2.4 Maintaining Consistency when dealing with dynamism 94
8.3 Transactional Layer 98
8.3.1 Transactional Layer API 98
8.3.2 Mutual Exclusion Mechanism 100
8.3.3 Transactional Mechanism 108
8.4 Banking Layer 110
8.4.1 Transaction Commands Component 110
8.4.2 Account Management Component 111
8.4.3 Security Management Component 112
8.5 Implementation details 116
8.5.1 Message Dispatching Component 116
8.5.2 Synchronous Communications over asynchronous primitives 121

9 System Evaluation and Characterization 123
9.1 Execution environment 123
9.2 Evaluation 124
9.2.1 Transactional Mechanism Evaluation 124
9.2.2 Consistency Mechanism Evaluation 128
9.3 System comparison 130

10 Project Plan and Economic Evaluation 133

11 Conclusions and Future Work 137

III Appendixes 141

A Conventions for the notation of algorithms 143

B User Guide 145

B.1 Basic Configuration 145
B.2 Execution 147
B.2.1 Localhost Setup 147
B.2.2 Distributed Setup 150
B.2.3 Playing with it 150

C Glossary **153**

Bibliography **157**

List of Figures

1 Grid4All and Virtual Organizations overview . . . 3

1.1 Taxonomy on currency systems . . . 9
1.2 Peermint scheme . . . 22
1.3 Karma scheme . . . 23

2.1 Common API for Structured p2p overlay networks . . . 29
2.2 Basic DHT structure . . . 31
2.3 Replication techniques . . . 32
2.4 Replica placement: Successor list disadvantage . . . 35
2.5 Replica placement: Symmetric replication approach . . . 36
2.6 Replica placement: Symmetric replication disadvantage . . . 37
2.7 Pastry: Plaxton-like routing system . . . 39
2.8 Distributed k-ary System: k-ary routing system . . . 41

6.1 Actors in the CMS System . . . 67

7.1 Grid4All Market Place architecture . . . 76
7.2 Payment Module logical view . . . 77
7.3 Currency Management System deployment view . . . 79
7.4 Currency Management System architecture . . . 80

8.1 Currency Management System layered view . . . 84
8.2 KBR Component Interface . . . 85
8.3 Mutable Consistent Component Interface . . . 87
8.4 Two different approaches for identifier assignment . . . 90
8.5 Basic Mutable Layer Communication Protocol . . . 95
8.6 Transactional Component Interface . . . 99

8.7 Transactional objects stored 101
8.8 TransactionManager Class Diagram 108
8.9 CMS Banking Layer Commands 111
8.10 Account and Receipts Class Diagram 112
8.11 Call to Command Sequence Diagram 113
8.12 Transaction execution example 114
8.13 Transaction Transfer of Funds logic 115
8.14 Message Dispatcher Class Diagram 117
8.15 Message Dispatcher Sequence Diagram 118
8.16 Message Dispatcher Reconfiguration Sequence Diagram 119
8.17 Message Dispatcher Serializer Sequence Diagram 120
8.18 Synchronous Communication Class Diagram 121
8.19 Synchronous Communication Sequence Diagram 122

9.1 Response Time vs Requests rate Comparison 126
9.2 Response Time vs Requests rate zoom Comparison 127
9.3 Throughput vs Requests Rate 127
9.4 Throughput vs Requests Rate Zoom 128
9.5 Probability of recovering stale data Comparison 129
9.6 Interval Recovery Comparison 130

10.1 Task duration and costs 134
10.2 Project Planification Diagram 134

B.1 Simple Web Interface of the Mutable Consistent DHT Layer 149

List of Algorithms

8.1 Create Object Algorithm 92
8.2 Update Object Algorithm 93
8.3 Query Object Algorithm 94
8.4 Interval Reconfiguration Mechanism to deal with node failures 97
8.5 Lock Object Algorithm 103
8.6 Unlock Object Algorithm 104
8.7 Commit Object Algorithm 105
8.8 General Transaction Skeleton 110
A.1 Algorithm Example 144

List of Tables

1.1 Comparison of Currency systems 21

2.1 Comparison of Structured Overlay Networks 42

3.1 Comparison of DHT Consistency mechanisms 49

4.1 Comparison of Mutual Exclusion over DHTs 56

6.1 Banking Service API Specification 68

10.1 Costs by roles 134

B.1 Configuration properties 146

B.2 Mandatory properties for building the overlay 150

Preface

Introduction

The necessity of greater computational resources by current enterprise and scientific organizations is increasing nowadays. This fact triggered what we currently know as *The Grid*. Foster et al.[1, 2] firstly defined it as a system which *coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service to users*. The general idea behind this definition is the creation of a decentralized system within which different organizations could share their computational resources to acquire a computational capacity impossible to achieve on their own.

One of the current main issues is the resource allocation problem amongst several users achieving high utilization rate without compromising user satisfaction. One mechanism which is currently gaining ground is the use of economic models in the context of resource allocation within the Grid, basically, due to:

- being an incentive to organizations for sharing their resources.
- being an auto-regulation mechanism for using resources based on the demand and supply law.
- being a mechanism to measure the importance (also called *utility*) of user's executed jobs.

As long as it is necessary some kind of transfer of rights mechanism, this economic model arises the issue of defining a series of protocols which enables the system to carry out the previously mentioned transfer of rights or, more commonly, *payments*.

Within this context, the Grid4All [3] European project has been created with the main objective of *democratize* the Grid to bring over it to little organizations and/or individuals without having to individually invest and manage computing and IT resources.

For that purpose, a framework for the development of Grid applications is defined with the help of which organization's users access IT resources through a market when their own resources turn out to be limited. This resource market is built upon the resources provided by different organizations taking part of the Grid.

Context

Basically, the world envisaged by Grid4All is builded upon some organizations which desire to share and consume resources to and from other organizations to fulfill their purposes (i.e increase computing and storage capacity).

This way, each organization (called Virtual Organization and its acronym VO) integrates services and resources accross distributed and dynamic organizations to allow service and resource sharing when cooperating on the realization of joint goals.

If the VO has not enough resources to carry out its objectives due to resource scarcity, it asks the market (See Figure 1) in order to allocate extra resources needed by the VO. Within this context, Grid4All provides the Market Framework which is in charge of defining different resource acquisition policies within the market itself.

So it is necessary to define mechanisms and protocols which enables the system to perform right transfers (*payments*) in order to maintain the accounting of which resources have been used by each VO and its associated price.

Goals and Scope

Once introduced the context upon which this project is stated, we define the main objectives as to:

Survey of different resource sharing accounting mechanisms : or, in other words, *payments*. The objective is to survey current existing solutions related with carrying out payments and currency management used for resource sharing and resource right exchange mechanisms.

Define specific Grid4All requirements : specify and select concret necessities in the context of currency management within Grid4All.

Design and implement a prototype : once specified the requirements, desing and implement a prototype of the basic components based upon those requirements in order to validate the system.

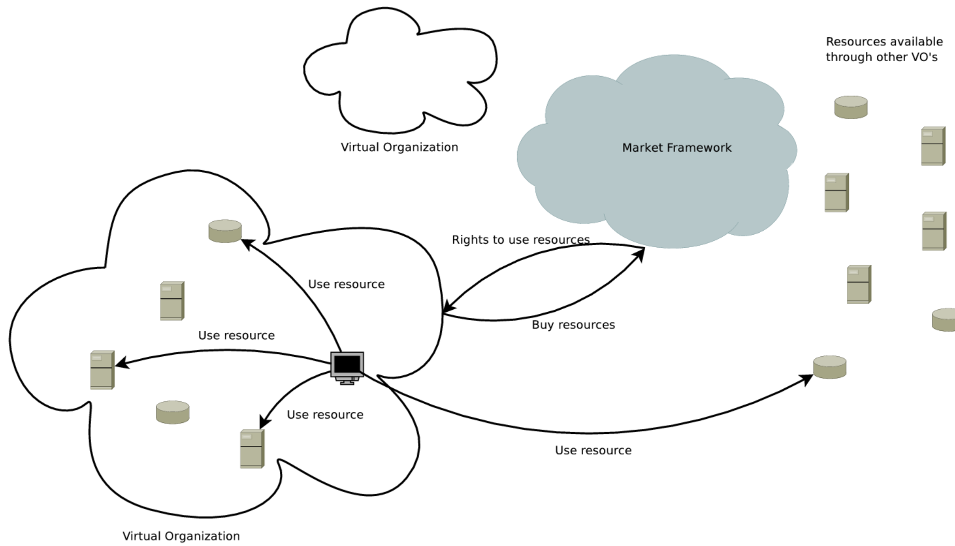


Figure 1: Grid4All and Virtual Organizations simple scheme. This figure shows how a user within a VO uses the VO available resources. Only if they are not enough to fulfill user requirements, the VO management framework will acquire more resources through the market.

Evaluate and characterize the system : validate the correct behaviour of our prototype as well as characterize its performance.

Project Overview

The starting point within Grid4All was the creation of a *Virtual Currency* which would enable the system to perform payments between clients and providers without having to rely on an external payment component outside the Grid4All infrastructure such as credit entities (credit cards through real banks) or PayPal. Both cases imposes a tax every time a payment is carried out.

As we will see through the rest of this document (and more concretely on Part II), one of the mandatory components for carrying out such transactions is a banking service which manages different user accounts within Grid4All.

This banking service should be reliable, fault-tolerant and scalable in the number of transactions it must support. For this reason, we have decided to implement it on top of a Distributed Hash Table (a sort of distributed storage based upon a structured peer-to-peer overlay network) which enables the system to take profit of several desirable properties for a distributed system such as its fault-tolerance, its inherent scalability as well as its self-management.

Nevertheless, operations such as transfer credit from one account to another one involve other requirements such as transactional semantics (ACID) used by current databases. For example,

transfer credits from one account to another implies modifying both accounts or, in case of failure, none at all.

This way, part of this project will deal with providing stronger guarantees when dealing with replicated data as well as providing transactional semantics by enhancing current DHTs implementations taking into account inherent dynamism within p2p networks.

To sum up, the final solution proposed by this project will be to develop a reliable distributed banking service prototype for the Grid.

Document organization

This document is organized as follows:

Part I: Concepts and Survey of Related Work. We begin by defining and surveying several concepts of currency related issues. For this reason, we provide a taxonomy of payment systems and provide a state of the art description on this area. Thereafter, we define DHTs related information such as their structure or their data management related issues (consistency and concurrency). Although this work was reported after the requirements of our system were defined, we have decided to introduce it within this part for the sake of clarity and organization.

Part II: The Currency Management System. Within this part we define the requirements and architecture of our system. Moreover, we provide a detailed view on the design and implementation decisions taken. Finally, we evaluate and characterize the performance of our decisions in order to learn from our prototype for future iterations. We finally provide project management issues such as the economical evaluation and project plan.

Part III: Appendixes. To facilitate reading this document, we provide a short glossary and conventions for the notation of algorithms used through Part II. We also provide a short guide to install and use our system in case the reader wishes to test it.

Part I

Concepts and Survey of Related Work

Throughout this part, we will introduce several concepts related to our work which will help the reader to follow the decisions taken and made a clear idea of the context this project is part of. Thus, Chapter 1 presents a taxonomy and a survey of different currency systems existing to date. The rest of chapters are DHT related issues. We begin with Chapter 2 by surveying and comparing different DHT solutions as well as the basic mechanisms on which they rely. We continue in Chapter 3 presenting different mechanisms to provide data consistency over DHTs and, finally, Chapter 4 introduces protocols to ensure mutual exclusion using DHTs as a substrate.

Chapter 1

Virtual Currency management and payment protocols

1.1 Introduction

A currency, as defined by the Wikipedia is a *unit of exchange, facilitating the transfer of goods and services. It is a form of money, where money is defined as a medium of exchange (rather than a store of value).* Therefore one function of currencies is facilitation and regulation since each country or region with a currency has some institutions (monetary authority) exerting control over the amount of circulating currency (the central bank or the ministry of finance).

Also from Wikipedia: *The origin of currency is the creation of a circulating medium of exchange based on a unit of account which quickly becomes a store of value*. Therefore currency can serve as an element to assign value or generic capacity to consumers, to facilitate and delegate actions/uses, to account for usage of services and resources, and to support the regulation of a system by enabling or limiting the generic capacity of consumers in respect to the capacity of providers and also limiting the storage of value.

The virtual currency has arisen as a necessity to apply economic policies in the scope of the regulation in distributed systems. Every system based on economic models has defined some type of currency in an explicit or implicit way. Currency definition is based in the concrete necessities of the system and therefore there is a great variety of these mechanisms.

In these systems, to guarantee the equitable distribution of the resources available, some type of bartering is made. For example, SHARP [4, 5] implements a policy of interchange of rights to use certain resources during a certain period of time. The problem with bartering is the complexity at

the time of materializing the exchange, due to the difficulty of matching different supplies. There is a tendency to define a generic entity as the base for the exchange. This allows more flexible goods representation and to make the economy most dynamic, such as it has happened with the real commerce throughout history. The properties more interesting that currencies have are:

Common unit of accounting: The users have a common unit accepted by every user in the system to value the goods traded in it.

Purchase power: The users decide how to use these currencies. Hence, they have a mechanism to prioritize the use of different goods based on their necessities.

This wide currency notion can be implemented in different ways based on different properties that the currency should has. The design decisions are bounded to the trade-off between security, scalability and availability of the system.

Although this study is focused in the technological characteristics of the payment systems, for a payment system to be succesfull, its design must consider the point of view of the end user who will use it. This way, one will have to consider properties such as the applicability (there are enough users and providers to match their needs), ease of use, convertibility (capacity to change the virtual currency to real currency), etc. Many of these characteristics can be found in [6].

Figure 1.1 shows different taxonomies, each of which explained in on subsequent sections, in which currency systems can be divided. So this chapter describes the concepts, uses and alternatives using a virtual currency and surveys the most relevant systems describing some kind of virtual currency to achieve their purpouse.

1.2 Currency Market

We can observe two types of currency markets: open and closed market. In both cases there will be problems related to the value of the currencies, such as inflation (maintained and generalized increase of the prices), deflation (opposite to inflation), etc.

1.2.1 Closed Currency Market

Currencies are created by a central organization (the minter). These currencies belong to the system and they cannot be replaced by outer currencies. Their only one objective is to acquire goods and services within the system where the currency is valid.

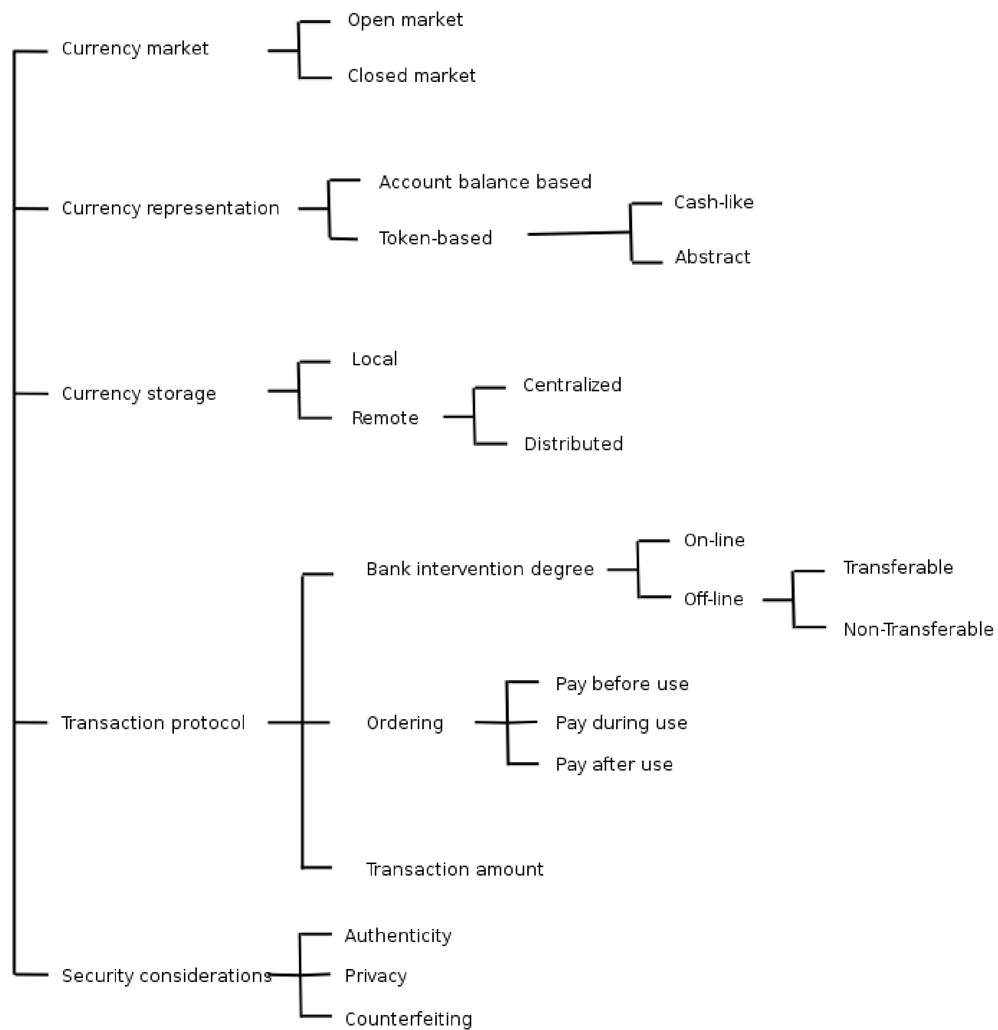


Figure 1.1: Different taxonomies in which currency systems can be organized. Notice that each taxonomy is not mutual exclusive with the rest

In these systems the excessive currency saving is a problem since the money in circulation diminishes and therefore the prices diminish due to the increasing value of the currency (deflation). Another problem related to the saving is that a user can save arbitrary currency for future spending. Once he decides to spend it, he can monopolize all the market. In order to solve this problem, we could think about currencies that have a certain time of validity during which the owner can use it to consume. Passed this time the currency no longer has validity. This measurement stimulates the consumption and limits the capacity of saving of the users.

The central bank must maintain the control on minted currencies to be able to apply monetary policies in the system, that is to say, increasing the number of currencies in case of deflation (to diminish the value of it currencies) or diminishing its number in case of inflation (increasing the value of the currency).

Most of systems use currencies to control the power of consumption of the different users. Hence, the exchange between virtual currency and real currency is not addressed in these systems. We can consider this kind of currency market as closed (Tycoon [7], Peermart [8, 9], Bellagio [10], etc.)

1.2.2 Open Currency Market

The currencies are created by a central organization as in the case of the closed market. In this case, these currencies are exchangeable by real currency outside the economic system (for example: euros, dollars, etc.)

Users purchase these virtual currencies with real money. They can use them within the economic system to acquire resources or services.

The problem with these systems is the capacity of a user "to put" too much virtual money in the system with which he acquires a great power to acquire goods and services, monopolizing the market. A possible solution could be to limit the acquisition of currencies by real money to avoid that a user has the capacity to saturate the system. That is, a user can only withdraw a limited amount of currency during a certain period of time (for example, 100 euros per month). As well as to limit the life of currencies as in the previous case to limit saving capacity.

There are systems that use virtual currencies to facilitate the interchange of real money to the users. Second Live, PayPal, e-Gold, BidPay, etc. are examples of them. These systems do not have the issues arised by the creation of an economic system since they serve solely like intermediaries to make payments between users.

1.3 Multiplicity of currencies

Taking into account the definition of currency provided in the first section (*medium of exchange*) we can consider that a currency has a region in which is the dominant medium of exchange. That is, in a specific zone there is a dominant currency with which trade goods. For example, in Europe there is the Euro, in the United States is the Dollar, etc.

An economic system willing to trade goods using some kind of currency should take into account that reducing the number of dominant currencies in a single common market (as the European Union) facilitates the trading in a wider zone (*currency zone*). Therefore, the possibility for matching supply and demand is higher without incurring in the problems with exchange between one currency to another.

Despite that, to facilitate trade between currency zones there are exchange rates (i.e. prices at which currencies can be exchanged against each other), that are defined by certain agents on the market specialized on that purpose and regulated by market institutions.

We can find some examples of systems implementing exchange between different currencies in a single system such as PayPal or e-Gold. Despite that they describe its service as a way to exchange real money via internet so they don't define different virtual currencies.

Other systems implementing a currency system (Ppay [11], Tycoon [7], Karma[12] and Off-line Karma[13], PeerMint [8], SecondLive, World of Warcraft, etc.) define a single currency inside their systems to trade with.

Taking into account the previous considerations, we can stat that a system willing to trade goods using some kind of currency should take into account that using a single currency will simplify its currency management as well as increase the number of users willing to trade with this currency.

In order to decentralize as much as possible the currency management, once every system has defined its own currency it would born new entities providing the exchange service between different currencies in different systems.

1.4 Currency representation

We can define two basic abstractions accepted by almost all existing literature to represent the value of this currency depending on how this currency is represented: account balance based and token based.

These two basic representations must be interpreted as a physical representation of the currency,

the way they are represented. That is to say: an account balance is a number stored somewhere representing the value of the goods a user owns; a token is an object representing a value unit and it is the minimal unit of representing a good (as an euro cent in the real world).

1.4.1 Account balance based

Every peer inside an economic system maintains an account with a bank by the total value of the goods that they own. In order to make the payment of a service, a transaction is made through the bank from the customer account to the provider account. This solution simplifies the management of the transactions between client and supplier. On the other hand, the currency representation doesn't have a physical object which the user can control (it is a number). Hence, they need a great confidence on the organization which is responsible for the account operations (add, subtract, etc.).

As we will see, we must consider that a centralized implementation of this bank (Tycoon [7], PopCorn Market [14], GridBank [15]) can arise problems with its availability due to the load of the central server. However, a distributed implementation along the users of the system (Karma [12], PeerMint [8]) must take into account the presence of malicious users who could influence maliciously in the correct operation of the system. The system relies in the consensus of several peers to overcome this reliability issue, with the consequential overhead due to the transmission of messages. This kind of representation need to maintain certain properties referring to security issues. The system should provide certain guarantees (see Section 1.7) when the account balance is modified in order to avoid falsification of payments. Depending on the *bank* implementation, it could be done by means of trusting the bank (in the centralized implementation) or by means of consensus of several peers (in the distributed implementation).

1.4.2 Token based

Each peer of the system has an amount of tokens proportional to the value of the goods that they own. In order to make the payment, the client must transfer a certain amount of tokens to the supplier by the value of the service. It is the most similar representation of what we know as cash.

Tokens must have certain properties for their correct management: uniqueness in the system (cannot co-exist two equal tokens); non-repudation (the possession of a token assures the possession of a certain value and the whole system must accept it as valid); non-forgeable (the users cannot create tokens, these single can be minted by a minting entity to which every user rely on).

We can find two main tendencies at the time of defining this kind of currency:

Cash representation: they try to construct a currency system as equivalent as possible to cash. In order to obtain it, they try to maintain properties such as unlinkability and/or untraceability, properties that cash inherently possess. These two properties are known jointly as anonymity. Systems that implement this type of representation are [16][17][18][19].

Abstract token representation: although the token has an associated value, it does not address to maintain the characteristics previously mentioned. They are designed to manage the resource consumption made by users and to avoid the free-riding problem. In these systems we can find PPay [11], CPay [20] or the work of Liebau et all [21].

As each token is the minimal unit of payment, there is an overhead associated with a transaction since each token must be digitally signed, maintain a history of the transactions, etc. This is due to some security issues, explained in Section 1.7, such as detect double-spending, the non-forgeability property, etc. In a market environment where the prices are not fixed and their variance is very high, the overhead generated by the transactions is not inestimable.

1.5 Currency storage

There are different ways to classify the currency management depending on where the currencies are stored. This classification will depend on the use case scenario as well as which currency representation is used (previous section). It is important to note that not all the representations introduced in the previous section could be stored using whatever form of storage explained in this section. We can define the currency by the following way: local, remote centralized and remote distributed.

1.5.1 Local

Each user is the one in charge of storing his currencies in a local device (hard disk, usb, local database, etc.), like a wallet. This way, each user is in charge of his accounts and decides when or with who do a transaction, without depending on a third party organization. This single abstraction has sense only if they store tokens because in an account balance based system, the user could vary his budget arbitrarily unless there is a trusted hardware which does the increase/decrease operations. Most of the systems implementing token representation store them locally.

1.5.2 Remote

The currencies are stored remotely to delegate their management to another organization inside the system. This organization is the one in charge of making transactions between client and supplier, and to maintain the balance of currencies updated. This decision can simplify the management of the security of the system, increase the degree of availability of currencies, facilitate the mobility of users, etc. Depending on the decisions such as decentralization, we can take into account two types of remote storage:

Centralized: the entity in charge to manage currencies is unique in the system and is in charge of controlling the correct behaviour of the users. This entity is often called bank (or broker). The security of the transactions only depends on correct behaviour of the bank. The protocols associated to the creation of accounts and the transactions have a high component of security such as authentication that can be solved by means of public key cryptography. Although the bank idea is a simplification for security issues due to the existence of a single trust entity, it is important to note that it becomes the bottleneck of the system.

Systems such as Tycoon [7] and PopCorn Market [14] follow this approach and both argue that the scalability of its system depends to a great extent on the load of the bank. And it could be very high in a system where the number of transactions can increase while the number of users increase. If the number of transactions are foreseable, this solution is ideal due to its simplicity as well as security.

Distributed: in this case, the entity in charge of the management of the transactions is distributed between the users who form the system. Its main objective is to reduce the load of the bank previously mentioned. Within this category we can find many points of view depending on the degree of decentralization and security that is desired and required for the system.

PeerMint [8] and Karma [12] implement a distributed bank where each peer (account holder) is responsible for the account of another peer (account owner). In order to receive certain degree of confidence in every transaction, each account holder is replicated depending on a replication factor which can be understood as a security parameter. In order to carry out a transaction, it must achieve a wide consensus through all the replicas to isolate the users who do not act honestly.

Other systems such as PPay [11] and CPay [20] implements systems based on tokens where each peer (token owner) is responsible of a certain number of tokens. The token owner is responsible of avoiding malicious behaviuor such as double-spending of tokens, counterfeiting, etc.

These distributed implementations assure a certain degree of confidence in the correct operation of the transactions. That is, in the presence of a high percentage of malicious users, the reliability of the system diminishes extremely. The problem of these systems is to achieve a reasonable degree of confidence. If the confidence must be high, the degree of replication must be high too, with the consequent load that takes place in the system due to the high number of messages needed to achieve a consensus between users. Although the centralized systems simplify the management of the transactions and assure a high degree of confidence, their scalability is their weaker point. If the main objective of the currency management is to allocate resources efficiently without a single point of failure (i.e p2p Grid) a distributed implementation seems more suitable in exchange for the commitment between reliability and load due to the replication in the system.

1.6 Transaction Protocol

The transaction protocol is defined in the system to carry out the exchange of goods by means of currency exchange in the system. Three main actors interact in every transaction: client (payer), supplier (payee) and the bank. As well as the actors, we can define three steps during the transaction protocol: withdraw, exchange of currency (transaction), deposit.

1.6.1 Bank intervention

We can classify transaction protocols depending on the degree of intervention of the bank.

1.6.1.1 Off-line protocols

The off-line transactions do not imply any contact with the bank (trusted authority). Only customer and provider are involved in the transaction. The most evident problem of off-line payment is the difficulty to prevent clients to use more money that they have, that is, to use the same token more than once (double-spending). This solution considerably diminishes the load of the central bank making the system more scalable. Despite that, it is impossible to detect the fraud by multiple spending during the transaction without the existence of a specific hardware which controls that (such as prepaid cards). A possible solution to this problem is to detect the problem after the fraud has been committed (after the fact or *lazy resolution*) instead of preventing it.

Within the off-line protocols, we can differentiate between those tokens that can be used more than once or those that after making a payment must be returned to the bank for clearing (renewed):

Transferable tokens: this property tries to diminish to the maximum the load of the central bank. It allows that the received token by a payment of a service can as well be spent for another payment without having to deposit it before. This property is especially necessary in p2p scenarios where the suppliers are clients as well as providers. Currency systems such as [18] implement this solution at the cost of delaying the detection of the multiple spending.

Non-transferable tokens: although partners continue to maintain the desired independence with respect to the central bank during the transaction, tokens received by the supplier must be deposited in the bank for clearing. In scenarios where clients and suppliers are different entities, it is a good election since once the suppliers receive the payments, the life of the currency finalize and can be deposited in the bank for their real economic retribution. That is, the supplier does not need to re-spend the received currencies.

1.6.1.2 On-line protocols

A transaction online implies the intervention of the bank in each one of the transactions. This solution is considered safer and simple than the off-line transactions since all the work is made in the bank. This way the fraudulent use can be prevented since the bank must accept the exchange being done. Systems such as [16][17] maintains the anonymity of the users who participate in a transaction in exchange for making the transaction online or nontransferable. There is a great tradeoff between preventing the fraud and making the protocol off-line.

1.6.2 Ordering in transactions

When a transaction is going to be made, one of the arising issues is when the delivering ordering of good is done. In this sense, we can define two basic operations along the transaction: *pay* and *acquire*. Payment is the instant where clients fund providers account by means of a *payment protocol*.

Acquiring a service/resource is the instant where the client executes/uses/acquires the goods negotiated. We should notice that, in general, these two operations are processed by means of different protocols (one for payment, one for delivering goods).

Serializing these two basic operations makes difficult to guarantee that nor clients neither providers do not cheat (consumers could pay but the service could not be obtained or service could be delivered but consumers could no pay for them).

Making these two operations atomically, that is, a client pays if and only if the good is delivered, is difficult to achieve (if not impossible) if we take into account the services and resources traded

in a grid market (cpu, memory, storage, video processing, etc.).

One example of atomic transactions are those ones used by NetBill [22, 23]. It assures atomicity only for specific kinds of goods such as webpages which can be encrypted before delivering the goods to client waiting for its payment. Furthermore, achieving these guarantees are constrained by the existence of a trusted central server which makes the system scalability very poor.

Since we cannot guarantee atomicity when transacting Grid goods, depending on the order of these two operations we can identify different protocols with which clients and providers have to agree during negotiation: *pay before acquire, pay during acquire, pay after acquire*.

This kind of protocols arise a security issue regarding the correct ending of a transaction. Neither client nor supplier have the certainty that the transaction is going to finish correctly. It is the most faithful representation of the transactions that we can see in the real world.

Pre-payment: *pay before aquiring the goods*. The client does not have the absolute certainty that the paid services will be given to him. A reclamation mechanism is needed to give back to the client the given currencies.

Post-payment: *pay after acquiring the goods*. The supplier does not have the absolute certainty that the given service will be paid. It is needed, like before, a reclamation mechanism.

Pay as you go: *pay during acquiring the goods*. The client makes small payments to the supplier while he uses the service. It is the fair mechanism for both. If the client detects that the service is not the negotiated one the transaction can finalize and if the supplier detects that the client is not giving to him the intermediate payments can stop the service execution.This last transaction mechanism is ideal for goods which are used for a long time such as executing a data-mining applications which can take hours or days to conclude.

In order to allow other payment mechanisms such as subscription (a client do a payment for some amount regularly to a provider in order to use a specified service/resource), it is only needed primitives such as *payment* and *receive a payment*.

As long as we will not be able to implement dispute free transactions electronically, we should maintain a log of transactions to resolve disputes between traders. This logging facility is necessary for accounting and auditing a transaction in order to reconcile the potential different points of view of clients and suppliers.

1.6.3 Transaction amount

The decision on what system is going to be implemented depends to a great extent on the real value that the transactions represent. Most of the systems spend huge time performing public key cryptography operations. If the real value of a transaction is very low, maybe the cost of public key operations may exceed the value of the payment which in turn is economically inviable.

Making a subjective division the payment can be divided as:

Low-value transactions: also known in the literature as micropayments. We could approximately understand them like payments by quantities from several cents to a euro. Their decisions are based on that the public key cryptography is too expensive for payments of small quantity. These systems are based on reducing to the minimum expression the use of the public key cryptography, making more emphasis in the use of hash functions. One of the first systems in implementing transactions of low cost was PayWord [24] by Rivest and Shamir. Their system is based on the unidirectional properties of the hash functions and implement what they call hashchains. Many other systems [25][26][27][28] are based on such concepts. It is important to note that micropayments are strongly focused in electronic commerce such as payments for watching web pages or for accessing private documents.

Mid-high value transactions: they are payments of variable amounts where the real value of the transaction surpasses the micropayments. In this type of payments the security is important since the falsification of a transaction could arise to great losses. For that reason, the costs associated to the security in the transaction are high. For example, the use of public key cryptography, the use of a trusted entity to confirm each one of the transactions, the replication of the entities in charge to manage the balance of users in systems of distributed storage currency, etc.

1.7 Security Considerations

This section covers only a small part of security related issues concerning currency and its design and implementation. If the reader is interested in this aspect a more accurate explanation of different solutions can be found in [16][17][18][19].

1.7.1 Authenticity

We can define authenticity by means of two important aspects in the electronic commerce:

Authentication: process by which the user identifies itself unambiguously.

Integrity of the messages: process by which it is possible to verify that the messages have not been modified throughout the transport in a non secure network (Internet).

Both aspects of the authenticity can be fulfilled easily by means of public key cryptography methods (cyphering data and signing them).

Authenticity is necessary both for account and token based systems because it is needed a non-repudation mechanism for monitoring the actions of different users and to achieve an irrefutable proof of that a payment is done by some user.

1.7.2 Privacy

In advanced cryptographic currency systems there is the necessity of introducing certain properties exclusive to real cash nowadays. That is, they try to apply the same properties of cash to virtual currency.

In real cash systems, once a user has withdrawn money from his account, the money (coins, notes, etc.) has no relationship with the user account from which they have been withdrawn. That is, the bank has no information about which real coins the user has (the bank doesn't record the bills retired although they are identified by serial numbers). This property is known as *anonimity*.

On the other hand, with real cash we can make payments of different amounts to a provider without revealing user's identity with respect to to the central bank. That is, the bank is not able to reproduce the history of transactions made with a single coin. This property is known as *untraceability*.

These two properties allow maintaining the privacy in the payment system. In the world of electronic currencies these characteristics can be obtained by means of different cryptographic methods at the cost of increasing the complexity and cost of the transactions.

David Chaum was the first in mentioning the anonymity as a necessity for electronic currency to be successful. He introduced the concept of blind signatures used later to achieve anonimity in electronic currency or in electronic voting systems.

The sketch idea of blind signatures applied to electronic currencies is the following one. A user mints a coin and he modifies it with a random value called *blinding factor*. This process is called *blinding the coin*. The bank signs the currency, which will have a random aspect, discounting the value of the coin from his account. When it is given back to the user, he eliminates the *blinding factor*. From now on, the user has a valid signed coin which has no relationship with the

withdrawal made before. When some other user deposits this coin in the bank, it will not be able to relate it with the creation as long as it doesn't know the blinding factor used to blind the coin.

For an exhaustive explanation of the cryptographic characteristics of these systems we direct the reader to the article of Chaum and its DigiCash system [16].

This basic idea has several disadvantages such as the ignorance of the bank of what it is signing. It could become easy for fraudulent users to give a blinded coin to the bank for a value higher than the value the bank think it is signing.

So, relaying in this basic idea, many other systems have been arising that refine this original idea removing some of their disadvantages.

1.7.3 Counterfeiting

Throughout the document, it's mentioned that one of the most important issues to take into account in payment systems is counterfeiting. The system must prevent impersonation (every user should know with whom he is dealing) as well as prevent token forgery and multiple-spending of a single token.

In order to solve the impersonation issue, systems usually relies on a PKI (Public Key Infraestructure) where a Certification Authority (even it could be bank) issues certificates which bound an identity (user's identity) with a public key. In this way, a token signed by the bank cannot be forged as long as the private key of the minter (the bank) remains secret.

Here, we should take note that anonymity is one of the issues which make a payment system complex. As long as anonimity is a desired property, we cannot rely on a PKI in order to achieve the security properties mentioned before. The system must ensure anonymity to a user as well as detect double spending of a token, and must be able to recognize the responsible of the fraud. Such systems, where the identity of a user is only uncovered when a fraud is committed, has been arising since Chaum. They rely on different cryptographic methods such as *cut-and-choose, zero-knowledge-proofs or secret splitting*.

1.8 Survey and Comparison

In this section, a more indeep sight of some currency systems is presented paying special attention in the architectural and design decisions as well as the protocols used to perform the payments and transactions.

	Representation		Storage		Transaction Protocol	
	Account	**Token**	**Local**	**Remote**	**Bank intervention**	**Transaction value**
Peermint[8]	X			Distributed	On-line	Indiferent
Karma[12]	X			Distributed	On-line	Indiferent
Off-line Karma[13]		X		Distributed	On-line	Indiferent
PPay[11]		X		Distributed	Off-line	Indiferent
CPay[20]		X		Distributed	Off-line	Indiferent
GridBank[15]	X			Centralized	On-line	Indiferent
Tycoon Bank[7]	X			Centralized	On-line	Indiferent
PayWord[24]		X	X		Off-line	Low

Table 1.1: Comparison of Currency systems

In Table 1.1 it is shown a comparison between these systems in order to have a summary of the most interesting properties and how these systems solve the problem.

1.8.1 PeerMint

PeerMint [8] is an accounting scheme totally decentralized and secure. In this system, the bank role is distributed among the peers which conform the whole system. The key concept of its design is that it relies on untrusted components (peers) to carry out secure and reliable transactions.

It is an account based scheme and each peer is the responsible to update and manage a bunch of accounts. It relies on a PKI [29] to assign each peer of the network a public/private key pair in order to identify them uniquely.

Due to its key idea of a totally decentralized currency management architecure, PeerMint has to deal with unpredictable behaviour of peers such as *malicious peers*[1] or the *collusion among peers*[2]. Figure 1.2 shows the basic architectural scheme of PeerMint.

To have a robust accounting scheme and overcome previous mentioned problems, PeerMint uses *session peers* which are chosen hashing the *id's* of the two peer involved in the transaction. These session peers may change every transaction, hence the probability of coallition is diminished.

[1] Peers responsible for a bunch of accounts and change them arbitrarily without explicitly having been asked for a transfer of funds

[2] Coallition of peers which allows an account to be changed arbitrarily to overcome redundant information which could serve as a basis to discover this bad behaviour

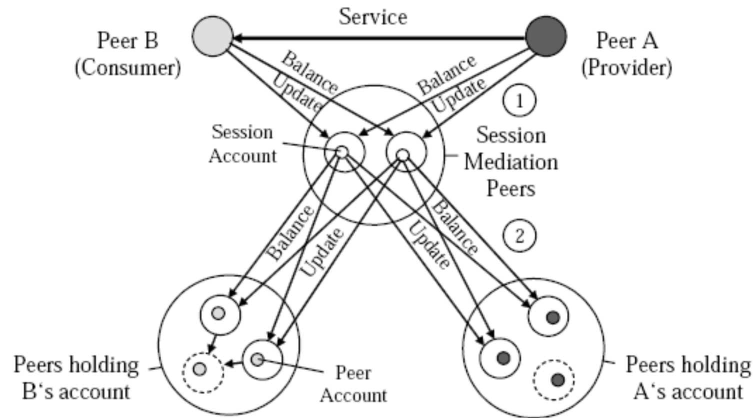


Figure 1.2: PeerMint basic scheme. Figure taken from [8]

Furthermore, each entity in the system (*account holder*[3] *and account session*[4]) is replicated to a factor of k. Hence, every modification of any account has to be performed by each replica and achieve a consensus.

This solution has the advantage of a completely descentralized architecture but it lacks of scalability due to the overhead generated by redundant messages sent to achieve consensus.

1.8.2 Karma

Karma [12] is an economic p2p framework which keeps track of the resource consumption and resource contribution of each participant. Each participant in the system owns an account represented by a single scalar value (*karma*). This account is managed by a set of replicated nodes (*bank set*) intended to resits malicious behaviour of the resorce provider, consumer or a fraction of the members of the bank set.

It is an account based scheme where each account is managed by a set of peers. Each peer is assigned a secure identifier and its account is assigned an identifier equal to $hash(nodeID)$. Thus the bank set which manages this account are the k closest nodes in the identifier space to the account's id.

Assume a node A and an account id $h(A)$. Each member of the bank set of $h(A)$ stores the amount of karma signed with A's private key, as well as a transaction log containing recent payments. Signing of the balance by A ensures that the value is tamper resistant. The transaction log acts as

[3]Peer responsible to managing peer accounts
[4]Peer responsible to carrying out the transaction between two accounts

proof of A's payments.

Karma deals with the problem of *inflation* and *deflation* by re-valuating periodically the outstanding karma so that the per-capita karma is maintained at a constant level. This re-valuation comes at a cost of $\theta(N^2)$ messages where N is the number of peers in the system.

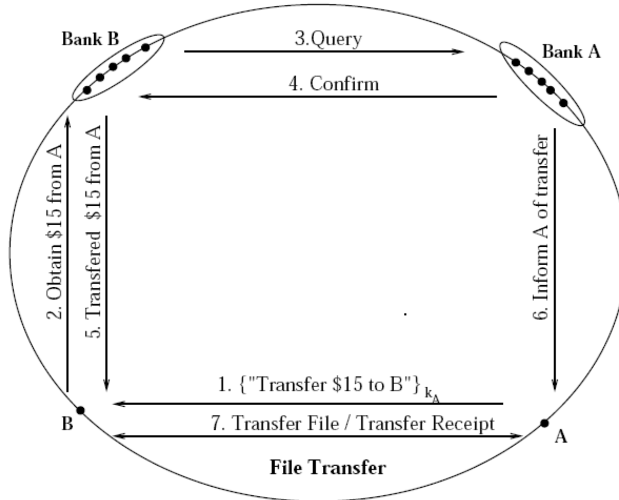


Figure 1.3: Karma basic scheme. Figure taken from [12]

As can be seen in Figure 1.3, the k members B's bank set must coordinate the transaction with the k members of the A's bank set leading to a message complexity of $\theta(K^2)$ due to all to all communication, introducing a high overhead for each transaction in order to avoid malicious nodes.

As in the case with PeerMint, this solution has the advantage of a completely descentralized architecture but it lacks of scalability due to the overhead generated by all two all comunications during transfer of funds and the re-valuation mechanism to avoid inflation and deflation.

Explicar Karma y decir que el protocolo de off-line karma es similar pero basado en tokens (y que los tokens van aumentando de tamaño a lo largo de su vida).

1.8.3 PPay and CPay

PPay [11] (a.k.a PeerPay) is a transferable token-based system. Its main idea is to delegate to the peers the burden of the bank's work (single server in charge of managing the account balance of each peer).

Assume a single bank which mints coins. Each peer of the system buys *coins* at this bank and from now on this peer is the *owner* of such a coin. To transfer the coin to another peer in order

to do a payment, instead of involving the bank server to do such a transaction, the owner of the coin transfer the ownership of the coin to the receiving peer. The receiving peer then is called the *holder* of the coin. Once the holder wants to spent this coin again, it contacts the owner in order to change the coin's ownership again to the new holder. If the owner of the coins goes off-line then the bank takes over the coin for future transfers.

This way, the bank is only involved in minting coins and in selling them. Their transferability is delegated to the peers. This scheme is efficient and scalable while peers are on-line for long periods of time but, if peers join and leave very often, the bank will be involved in almost every transfer so the burden of the work will be given back to the bank worsening its efficiency and scalability.

CPay [20] shares similaraties with the previous system but the way of choosing the owners of the coins. CPay only allows long-lived peers to own a coin and be responsible for transfering them. An incentive is given to those peers proportional to the work done. This way, CPay deals with the heterogeneity inherent to p2p systems.

1.8.4 GridBank and Tycoon Bank

Both systems are centralized banks in the whole sense of the word. Despite that, GridBank is extensible to whatever kind of payment currently (or futurely) deployed in e-commerce. It implements a resource usage mechanism by means of RURs (Resource Usage Record) which helps in the task of accounting what resources have been consumed by which users. The other way, Tycoon Bank has a simple protocol based on xml-rpc invocations to the server. Different transactions are serialized in order to avoid inconsistencies while transferring and, thus, provide ACID properties to its databse. It manages security by means of Public Key Criptography.

The scalability and efficiency of both systems is poor due to its centralized architecture and the cost of cyphering and decyphering messages. Despite that, both systems argue that their scalability is enough in order to carry out their purpose.

1.8.5 Hash Chains

This family of protocols are based on the idea that public key cryptography has an expensive computational cost and, therefore, the cost of performing such operations outweight the economical cost of a low-value transaction. In other words, the argue that the cost of counterfeiting a transaction outweight the transaction's value. A group of protocols implementing this idea are [24][25][27] [26].

The basic idea is to minimize the public key cryptography by using the efficiency provided by one way functions (a.k.a hash functions). Let H be a hash function (i.e MD5, SHA1, etc.) so it is easy to compute but difficult to revert. A user computes a $H - chain$ consisting of the values

$$x_0, x_1, x_2, ..., x_n$$

where

$$x_i = H(x_{i+1}) \text{ for } i = 0, 1, 2, ..., n - 1$$

Then it sends the root x_0 along with the signature of this token to the resource provider. To perform successive payments, the user sends the next consecutive hash value. The provider only has to verify that
$x_{i-1} = H(x_r)$
where x_{i-1} is the last verified payment and x_r is the payment to verify.

For the provider be able to withdraw those payments to the bank, it has only to send the last received payment x_i along with the signature of x_0. The bank only has to iterate i times H against x_i and verify that the value x_0 is equal to the signed value sent by the vendor.

This system resuces the cost of the transaccion. Despite that its flexibility is very low due to the impossibility to agregate payments from different clients as well as the reutilization of those hash chains. In p2p systems, the relationship between clients and provides might not be stable so the use of a different hash chains for different partners might have a high associated cost.

Chapter 2

Distributed Hash Tables

2.1 Introduction

A distributed has table is, as its name suggests, a hash table which is distributed among a set of cooperating computers, which are refered to as *nodes*. Just like a hash table, it contains key/value pairs, which are refered to as *items*. The main service provided by a DHT is the *lookup operation*, which returns the value associated with any given key. Moreover, a DHT also has operations for managing items, such as inserting and deleting items.

Because of limited storage/memory capacity and the cost of inserting and updating items, it is infeasible for each node to locally store every item. Therefore, each node is responsible for part of the items, which it stores locally.

This architecture enables DHTs to efficiently handle large amounts of data items. Furthermore, the number of cooperating nodes might be arbitrarily large (from a few nodes to many thousand or millions in theory).

In 2001 the first DHTs where developed and since then, several other algorithms developed to improve DHTs results have arised with similar properties but different implementations. These essential properties are:

Scalability in terms of:

- *Routing information*: the typical number of hops required and the size of the routing table is less or equal to $log(n)$ where n is the number of nodes.
- *Items dispersed uniformly*: each node stores an average of $\frac{d}{n}$ items where d is the number of items and n the number of nodes.

- *Support for dynamism*: a join/leave or a failure of a node, only implies the redistribution of the average storage of a node ($\frac{d}{n}$).

Self-managment in terms of:

- *Routing information*: when a node join/leave/fail, routing information is updated accordingly to reflect the event generated.
- *Items stored*: when a node join/leave/fail, the items stored in the DHT are redistributed automatically depending on the type of the event.
  - *In case of join*: the new node retrieves the items it is now responsible for
  - *In case of leave*: the old node pushes the items to the new responsible node
  - *In case of failure*: the new responsible node retrieves the items from the replicas maintained in the DHT.

Fault-tolerant: a DHT beeing fault-tolerant implies that lookups should be completed successfuly even if some nodes fail. This is typically achieved by replicating items as it will be explained later in Section 2.3. Hence failures can be tolerated to a certain degree as long as there are some replicas of the items on some alive nodes. For a complete description of what a fault tolerant system need to achieve see [30].

A key design issue is how to map different identifiers to some nodes and made it in some way that the system maintain its properties. Most of the DHTs were based on one of two ideas for mapping key identifiers to nodes:

Consistent hashing : is a hashing scheme such that the number of items needed to restore the state of a node is minimized when nodes are added or removed [31]. Consistent hashing tends to balance load, since each node receives roughly the same number of keys and involves relatively little movement of keys when nodes join or leave the network.

Plaxton mesh : is a scheme that enables efficient routing to the node responsible for an object, while requiring a relative small routing table [32]. The idea is basically, once each node has an identifier associated, route the message to the longest maching prefix of the routing table. For example, if a node has in its routing table the nodes A14B6 and A1357 and it has to forward a message to the id A1300 it will route the message through the node A1357 as the longest maching prefix is A13**.

Currently, a recent publication [33] argued that a common API for these kind of overlays (a.k.a structured p2p overlays) would *facilitate independent innovation in overlay protocols, services,*

and applications, to allow direct experimental comparisons, and to encourage application development by third parties. Moreover, it serves as a reference to classify different layers in the development of an application based on one of these structured p2p overlays. This way, an abstract representation of the different services provided by each layer of the DHT can be separeted driving the application developer to clearly design it independently of the underlying DHT infraestructure.

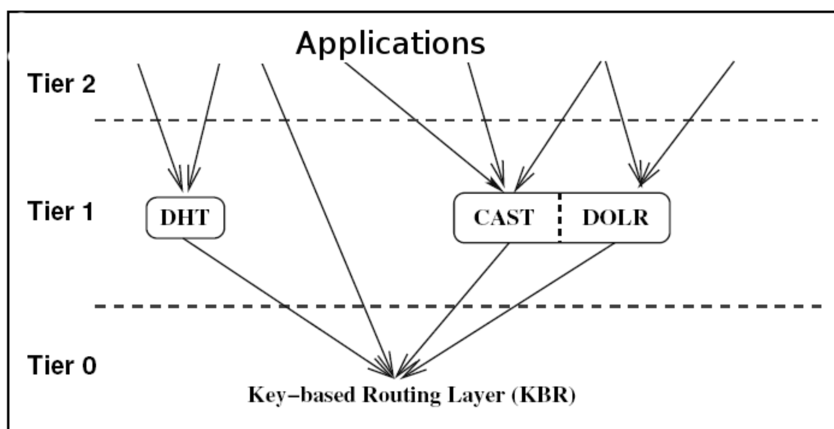


Figure 2.1: Common API for Structured p2p Overlay Networks

In Figure 2.1 from [33] there is the disctintion between the different layers. As long as this project will be developed arround the KBR Layer and the DHT Layer a brief description of these two components is as follows (for a complete description of the each layer and a simple API for each one see the previous cited article):

KBR Layer : provides a routing layer by which each node is able to send a message to an identifier. This message will be processed by the responsible node at the sending time for the given identifier. The message might run across several nodes depending on the protocol specific algorithms for routing.

DHT Layer : provides the same functionality as a traditional hashtable, by storing the mapping between a key and a value. This interface implements a simple store and retrieve functionality, where the value is always stored at a live node in the overlay to which the key is mapped by the KBR layer. Values can be objects of any type.

Once outlined the general idea and characteristics behind DHTs, a more deep sight is necessary in order to understand the solution proposed in Part II.

Section 2.2 make a more precise explanation of the internal structure of a DHT and how the nodes are organized to achieve the properties previously mentioned. Section 2.3 shows replication techniques used to maintain persistent data and provide a certain degree of fault-tolerance to the DHT. Section 2.4 makes a brief explanation of the algorithms used by three current DHTs.

2.2 DHTs structure

So far, this chapter has outlined several important properties that DHTs offer. Despite that, it is important to introduce the general structue and organization of the DHTs to understand the different approaches taken to address these properties.

Throughout this document, only *logarithmic state DHTs* are introduced taking into account the taxonomy presented in [34]. They are based on the idea that the minimum information necessary to lookup a key in $\theta(log(N))$ hops is $\theta(log(N))$, where N is the number of nodes in the overlay network. Three examples of this kind of DHTs are surveyed in Section 2.4.

Two basic components of a DHT are their overlay network topology and the routing information used to achieve the lookup operations. Although each DHT implementation differs substantially, some common characteristics are shared and introduced in the next subsections to provide a global view of how a DHT is organized and managed.

2.2.1 Overlay network topoly

Every structured overlay network makes use of an *identifier space* consisting on the integers 0,1,...,N-1 where N is a well-known fixed parameters representing the maximum number of identifiers that nodes may use.

Every node in a DHT has a unique identifier from the identifier space. Each node u has a pointer[1] to the first node following it clockwise on the identifier space (*Succ(u)*) as well as the first node preceding it (*Pred(u)*). The nodes therefore form a kind of distributed double-linked list that is sorted by the identifiers. Almost DHTs refer to this linked list as the *ring*.

Every identifier in the identifier space is under the responsibility of a node in the following way. The whole identifier space is partitioned into P intervals, where P is the current number of nodes in the system. Each node, n, is *responsible* for one interval. This way, a node is responsible for the interval consisting of all identifiers in the range $(pred, n]$ wher pred is the predecessor's identifer and n is its own identifier (See Figure 2.2 for a basic dht structure).

To provide the DHT a certain degree f of fault-tolerance in case a node fails, each node keeps track of its f closest nodes in the identifier space. This way, if a set of k consecutive nodes in the identifier space fail, the ring will be still connected. This list is usefull too to trunk the lookup process when a message is close enough to avoid making more hops than necessary. These list are

[1] A pointer is an entry with information about the node's identifier and its network address with which the communication can be established

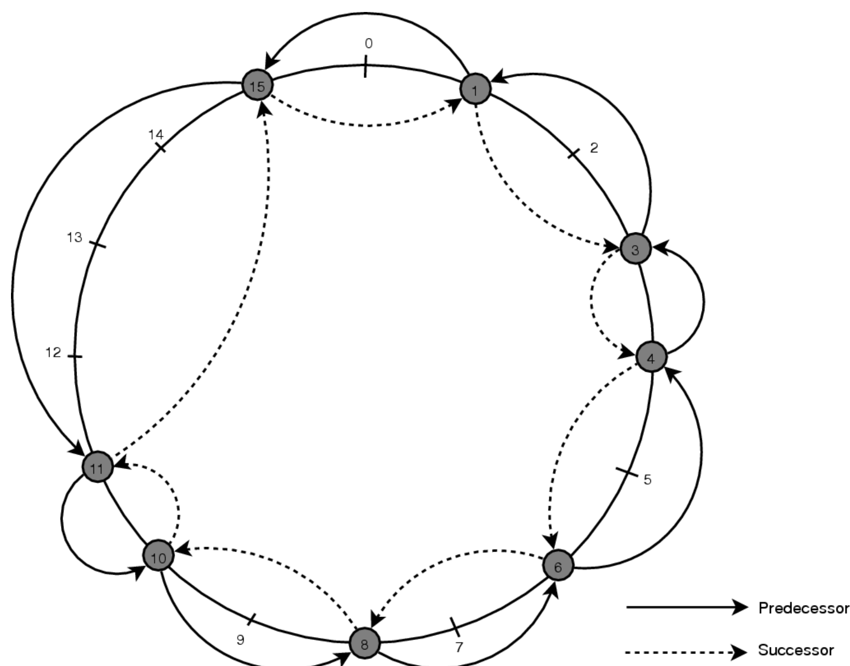


Figure 2.2: It is a DHT ring based topology as an overlay network. The identifier space is in the range [0, N-1] where N=16 in this case. Each node u has its own Pred(u) and Succ(u). All together form a double-linked list without beginning nor ending. It is shown how Node 15 is responsible for the identifiers between its predecessor and itself (12,13 and 14).

called *predecessor* and *successor* lists (a.k.a *leaf-set*).

2.2.2 Routing information

As important as the overlay network topology is how a message is routed to achieve the $\theta(log(N))$ complexity in the number of hops. So far, with the predecessor and successor pointers, a basic lookup algorithm could lead in the worst case to N hops to reach the responsible of a given identifier.

The first idea was to introduce some other routing information to make the path to the destination shorter. The same idea as the dichotomic search in linked lists could be applied to this ring. Each node keeps track of a node with which divide the identifier space by a half, another one by a quarter, and so on... This way, each hop jumps at least half of the identifier space remaining to arrive to the destination.

Depending on the definition of the identifier space and the routing protocol, the information stored in the routing table could change significantly as it will be shown in Section 2.4.

2.3 Replication schemes

2.3.1 Replication as a technique to achieve fault-tolerance

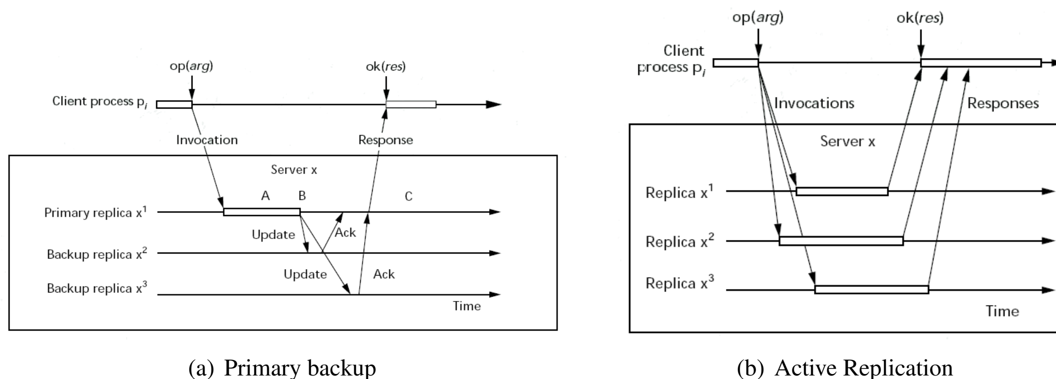
The key tecnique to actually achieve fault tolerance in distributed systems [30] is *redundancy* (a.k.a replication). The idea is to replicate processes (or components in a distributed system) and organize them in groups. The key property that this group of identical replicated components has is that when a message is sent to the group itself, all members of the group receive it. In this way, if one process in a group fails, hopefully some other process can take over for it [35].

The purpouse of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to complain about its internal structure.

There are two important issues concerning replication schemes when designing a fault-tolerant system.

Replication scheme : defines the way a group of processes could be logically organized and how the communication protocol is managed (when, how and to who a message is sent). These schemes are rather generic and system independant, hence they serve as a basis to construct fault-tolerant system.

Replica placement : defines the placement of different replicas to achieve certain properties (geographical locallity, load balancing, etc.). This decission is system dependant and specific to system protocols so it will be shown in Section 2.3.2 different replica placement schemes for DHTs.



(a) Primary backup (b) Active Replication

Figure 2.3: Two different approaches for replication

Depending on how the messages are sent and how, when and who updates the replica state, replication schemes can be divided into two main techniques as shown in Figure 2.3:

Primary-backup replication: This technique uses one special replica which plays a special role. It receives requests from clients, processes them, update its own state, send the state to backup replicas and waits until all backup replicas send a response. Once all backup replicas have been updated, the primary returns the response of the request to the client. If the primary replica fails, one of the backup replicas takes over from the group as a primary.

Active replication: Also called *state machine approach*, this technique considers all the replicas with the same role without the centralized management of the primary-backup approach. In this scenario it is the client who sends a messages to all the non failing nodes inside the replica group. This methodology requires that all nodes receive every message in the same order in order to carry out the message processing properly. The failure of one replica is transparent to the client as long as the client is responsible to wait for the response of one, all or some of the replicas.

Each replication technique has its own pros and cons. For example, in a primary-backup scheme the failure of the primary is not transparent to the client and therefore the client might perceive an increase in the latency waiting for the response due to the reissuing of the request. By the other way, in an active replication scheme the failure of a node is transparent. Nevertheless, it needs a total order in the reception of the messages in every replica in order to process the request consistently with the rest of the group. In this case, a primary-backup scheme supports an implicit order in the messages as long as the primary replica is the responsible to send the messages in order to the rest of the replicas.

2.3.2 Replication mechanisms in DHTs

In this section an overview of different replica placement mechanisms is introduced. Replica placement in DHTs can be defined as the mechanism by which a node elect other nodes to maintain a replica of the content beeing stored in the DHT. Replication over a DHT assume a replication factor of f, where f is the number of replicas that the DHT must maintain to achieve the desired degree of fault-tolerance.

The main mechanisms to choose replica placement can be devided in *neighborhood replication* or *identifier replication*.

2.3.2.1 Neighborhood replication

The idea is to elect the nodes which will conform the replica set from the neighborhood of a node. A neighborhood of a node can be understood as its successor list or its leaf set. These two

approaches are very similar and its applicability depends on the routing protocol of the underlying KBR Layer.

Leaf-set replication : stores a replica of an object with identifier *id* in its $\frac{f}{2}$ closest successors and its $\frac{f}{2}$ closest predecessors. This mechanism is used by Pastry [36].

Successor-list replication : stores a replica of an object with identifier *id* in its f closest successors of the item's identifier. This mechanism is used by Chord [37].

The reason for this difference is due to the routing protocol used in the underlying KBR Layer. If the routing always proceeds in clock-wise direction, the best approach is using successor-list replication, because the new responsible for a given identifier in case a node fails is its successor. A DHT using this mechanism could be Chord [37]. If the routing protocol can proceed in both clockwise and anti-clockwise direction, the best approach is using leaf-set replication, because the responsible to solve lookup operations in case a node fails could be its successor or predecessor depending on the direction taken by the message routing. This mechanism is used, for instance, by Pastry [38].

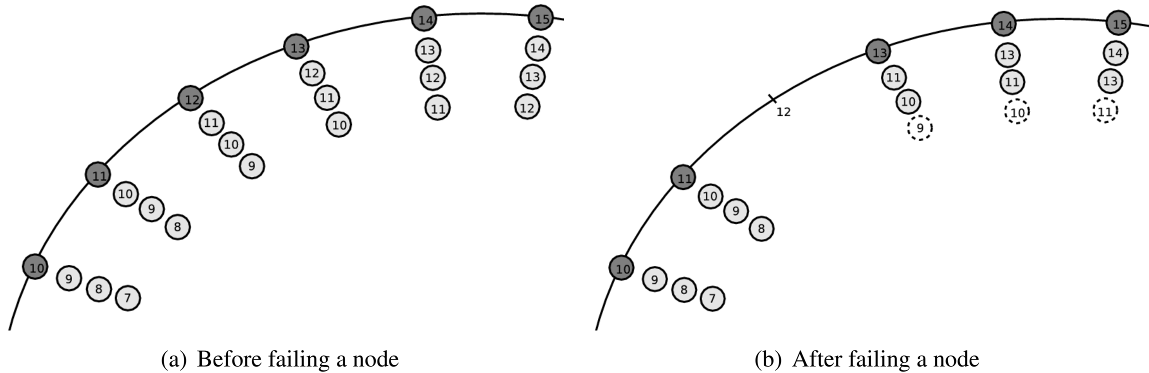
These solutions were proposed in order to fulfill a main purpose: replicate items stored in p's successor node, such that if p fails, lookups can be handled by its successor, since p's responsability is delegated to its successor automatically when p fails.

Nevertheless, these schemes have some disadvantages. The most import one is the necessity to exchange at least f messages *every time* a node joins or leaves. By definition, the f nodes inside the leaf-set (or successor list) of a node which leaves the system will belong to the leaf-set (or successor list) of a node which they where not in previously. Hence, a message to each one of these new neighbors has to be sent. Figure 2.4 shows an example.

2.3.2.2 Identifier replication

The consequence behind neighborhood replication is that the nodes are the subject of the replications. That is, each node replicates every item stored in its f predecessors, no matter how many items has each node. A range of identifiers very populated would lead to a saturation of the nodes which covers that range.

The idea behind identifier replication is, as its name suggests, replicate identifiers, not nodes. Hence, every identifier has a fixed set of replica identifiers selected by a specific mechanism. So every node responsible for each one of the replica identifiers will host the item as a replica.



(a) Before failing a node

(b) After failing a node

Figure 2.4: Assume a replication factor f of 4. For each node n (black ones in the figure) there is a predecessor pred and a successor succ. Each node n stores items in the range (pred, n] and replicates the items stored on its f predecessors. In 2.4(a) node 12 has stored its own items and the items from its f predecessors (9,10 and 11). In 2.4(b), node 12 has failed. Therefore, to maintain the replication degree of 4 some other nodes must take over them. In this example, nodes 13,14 and 15 replicates items from nodes 9,10 and 11. So f messages must be sent.

This approach has several advantages with respect to *neighborhood replication*. First, this approach enables the application to perform parallel lookups to any number of replicas concurrently. Therefore, a node can speed up the lookup process by picking the first response that arrives. Second, the availability of the content stored is rised up due to the different routes that concurrent lookups take. For example, when searching for the item *i* with identifier *k*, concurrent lookups are made to the different nodes which holds the replica identifiers for *k*, leading to mutually exclusive paths. If one path to a replica is broken (i.e. an intermediate node has failed) the rest of the paths are unaffected and might finish correctly.

Finally, join and leave operations have a message complexity of $\theta(1)$ reducing the $\theta(f)$ complecity of *neighborhood replication*. A node joining or leaving only requires to exchange data with its successor prior to joining or leaving. No other exchange of data items is required to restore replication degree.

Two different approaches to select the associated replica set of a given identifier are:

Multiple Hash Functions : it is based in using several hash functions for determinig replica placement. For a replication factor of f, f hash functions are needed and the replica set is obtained by hashing the identifier of the item to replicate with each one of the hash functions. This mechanism is implemented by CAN [39] and Tapestry [40].

Symmetric replication : the idea is to divide the identifier space into $\frac{N}{f}$ equivalence classes such that identifiers in the same equivalence class are all associated with each other. For example, in an overlay network with N=16 and a replication factor f of 4, the identifiers 0, 4, 8 and

12 belongs to the same equivalence class modulo f. This example is shown in Figure 2.5. This scheme was built specifically for DKS [41] but it is rather generic to be adopted by any DHT.

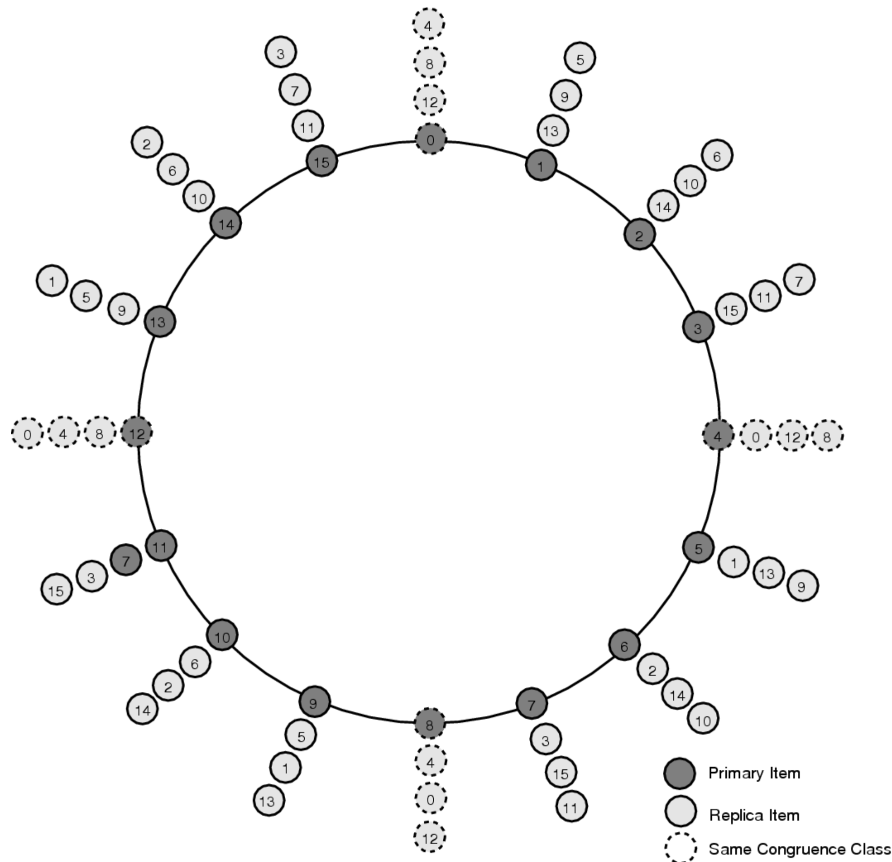


Figure 2.5: Assume a ring with 16 nodes and a replica factor f of 4. Identifiers in the same congruence class are associated. In other words, the responsible to store an item with identifier id will be the responsible to store a replica for the rest of identifier in the same congruence class as id. This leads to a symmetric balanced storage along the ring.

The main difference between these two mechanisms to find the replica identifier is its load balancing across the ring. While in symmetric replication each identifier (and therefore each item) is stored at fixed associated identifier and in a symmetric way along the ring, in the multiple hash functions approach the replicas for an item are dispersed along the ring depending on the result of each hash function, which makes it difficult to predict a fair load balancing across nodes.

Despite these schemes seems to solve the problem arised with *neighborhood replication*, they have its own drawbacks. In the case of *multiple hash functions mechanism*, the inverse of the hash function must be known to maintain the replication factor. It means that, as long as this requirement is impractical, the the replication degree is not restored once a node has failed. The

parallel *put* and *lookup* mechanism could solve this problem. Nevertheless, this option could lead to an item dissapearing from the system if it is not often updated.

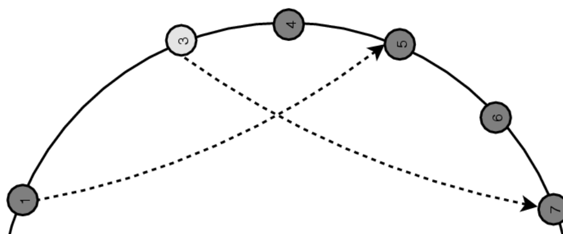


Figure 2.6: Based on Figure 2.5. The interval (1,3] is replicated by the nodes in the interval (5, 7]. If node 3 fails, the interval stored in this node must be restored from two different nodes (nodes 6 and 7).

In the case of *symmetric replication*, althought it is possible to restore the replication degree once a node fails, it could lead to a higher complexity. It means that the interval belonging to a failed node may be recovered from more than one alive node. Figure 2.6 shows this complex retrieving of items.

2.4 Survey and comparison

As outlined previously, DHT's are a massively scalable way to organize several nodes in an overlay network. In this section it will be shown how different DHTs are organized and managed taking into account the basic structure outlined in the previous section. For a larger and exhaustive explanation of each system see the referenced paper in each subsection as well as [42] and [34] for a good comparison. A summary table comparing these systems can be found in Table 2.1

We have selected four DHTs. Chord is one of the first implementations of a DHT and it is used as a reference due to its simplicity. Pastry and Tapestry are very similar and they are based on the plaxton mesh. Finally, DKS is a novel approach as a generalization of the Chord architecture but with a low-bandwidth topology and several algorithms which improves DHTs performance. Just to mention, DKS is being developed at the *Royal Institute of Technology (KTH)* and the *Swedish Institute of Computer Science (SICS)*, both of them partners of the european project Grid4All [3].

2.4.1 Chord

Chord [37] assumes a circular identifier space of size *N* and unique IDs are associated with both data items and nodes by means of a variant of the previously mentioned *consistent hashing*.

Each Chord node *u* maintains information about:

Successor and Predecessor : each node has its own $Pred(u)$ and $Succ(u)$ to maintain the ring connected.

Finger Table : in addition, a node keeps $M = \log_2(N)$ pointers called *fingers*. The set of fingers of node u is $F_u = (u, Succ(u + 2^{i-1})), 1 \leq i \leq M$, where the arithmetic is modulo N.

In order to route a given message to an *id*, the node forwards the message to the closest finger in clockwise direction of its finger table. This way, the routing protocol assures that the distance in the identifier space is decreased by a half after each hop leading to a $\theta(log_2(n))$ complexity in routing messages.

When handling with dynamism, that is peers joining and leaving, Chord uses a *stabilization protocol* running periodically in the background to update the successor pointers and the entries in the finger table. The correctness of the Chord protocol relies on the fact that each peer is aware of its successors. When peers fail, it is possible that a peer does not know its new successor and thus the routing could fail. The stabilization protocol assures that failures are corrected eventually.

This DHT was one of the first implementations of the *consistent hashing* approach previously introduced (Section 2.1) to map identifiers to nodes. Although in its first implementation, Chord didn't handle replication, in [37] argue that replicating items in the *successor list* could lead to an enough fault-tolerant storage as long as if a responsible node for a given identifier fails, the next responsible will be the first successor in the successor list of the failed node.

2.4.2 Pastry and Tapestry

Pastry[38] and Tapestry[40] makes use of Plaxton-like prefix routing to build a self-organizing ring. Both of them are very similar so it will be shown how Pastry is organized in order to have an idea of how a system with a Plaxton-like structure works.

Each peer in the overlay network has a unique 128-bit nodeId. This nodeId is assigned randomly when a node joins the system. Each data also has a 128-bit key. The data is stored in the node whose id is numerically closest to it key.

Each Pastry node maintains a routing table, a neighborhood set and a leaf set.

Routing table : Assuming a network consisting of N nodes, a node's routing table is organized into $log(N)$ rows with $2b - 1$ entries each row. The n-th row of the routing table contains the nodeIds and IP addresses of those nodes, whose nodeId shares the present node's nodeId in the first n digits but different in the $n + 1$ digit. If there are more than $2b - 1$ qualified nodes, the closest $2b - 1$ nodes will be selected, according to proximity metric.

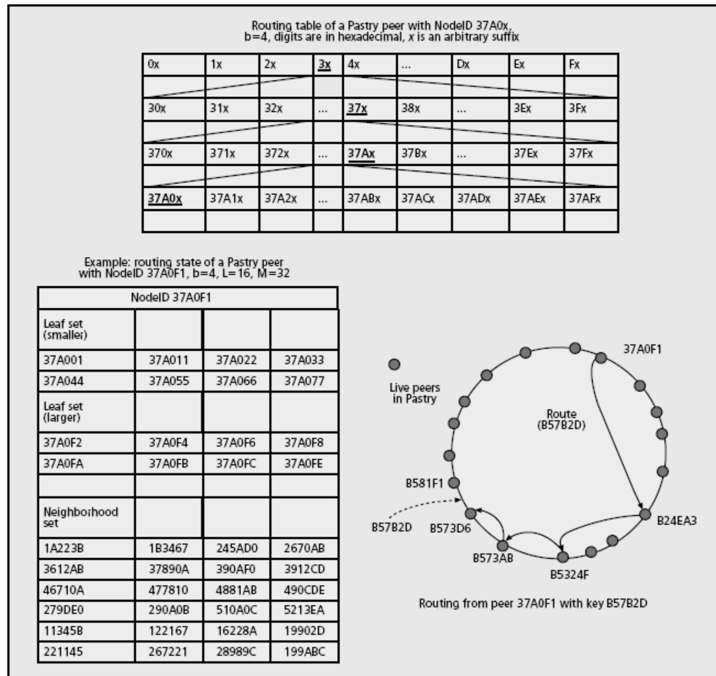


Figure 2.7: Figure from [42] shows a simple Pastry routing table, leaf set and neighborhood set of a Pastry Peer.

Neighborhood set : Neighborhood set contains the nodeIds and IP addresses of the nodes that are closest to the present node.

Leaf set : Leaf set contains the nodeIds and IP addresses of the half nodes with numerically closest larger nodeIds, and half nodes with numerically closest smaller nodeIds, relative to the present node's nodeId.

In order to route a given message, the node first checks to see if the key belongs to the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to the destination node, or in other words the node in the leaf set whose nodeId is closest to the key. If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit. In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the present node, and is numerically closer to the key than the present node's nodeId. In Figure 2.7, an example of a Patry node's routing table is shown.

The main difference between Pastry and Tapestry is its way of replicating the items. While Pastry makes use of the *leaf-set replication* mechanism, Tapestry makes use of the *multiple hash functions* approach.

2.4.3 Distributed *k*-ary System

Distributed k-ary System (or DKS for short) [43] began as a general abstraction that can be used to derive most of the existing DHT lookup services. In other words, the idea of k-ary search seemed to be general enough to derive several DHT-based algorithms. This way, most of the existing DHTs can be seen as instances of DKS.

Maintaining the idea of a general framework to develope and improve DHT algorithms, DKS introduce three parameters that specify some of its properties and can be adjusted to suit ones needs:

- ***N*** is the maximum number of nodes that can be part of a DKS. So far, the idea is similar with the basic idea of DHT explained in previous sections.
- ***k*** is the search-arity of the network. Searching for a value in the network can be modeled as a k-ary tree with the node that initiates the search as the root node. The higher the value of k, the less messages are needed to perform a search, but the bigger are the routing tables. k should be chosen according the formula $N = k^L, L \in N$, where L is called the number of levels in the search or the depth of the search tree. Figure 2.8 shows a ring with L of 3, a k of 4 and, thus, an N of 64.
- ***f*** is the fault-tolerance of the network. In any given DKS, predecessor and successor points to f consecutive nodes respectively. Thus, $f - 1$ consecutive nodes may fail simultaneously and the network will recover gracefully without loosing data. f should be chosen as a divisor of N.

In this context, it can be seen that Chord is an instance of the DKS where the k parameter is fixed to 2, and N and f the same as in the Chord definition.

Despite beginning as a framework for studying and analyzing different DHTs solutions, DKS growed up by itself with efficient algorithms that could be applied to other DHTs structures. Some special characteristics and algorithms which make DKS an efficient and special solution are:

Atomic Ring maintenance : provide algorithms to maintain a ring structure which guarantees atomic or consistent lookup results in the presence of joins and leaves, regardless of where the lookup is initiated. Put differently, it is guaranteed that lookup results will be the same as if no joins or leaves took place (a.k.a *lookup consistency*[2]). This is a strong guarantee on top of which more robust and hard semantics can be build.

[2] At any time, there is only one node responsible for a given identifier

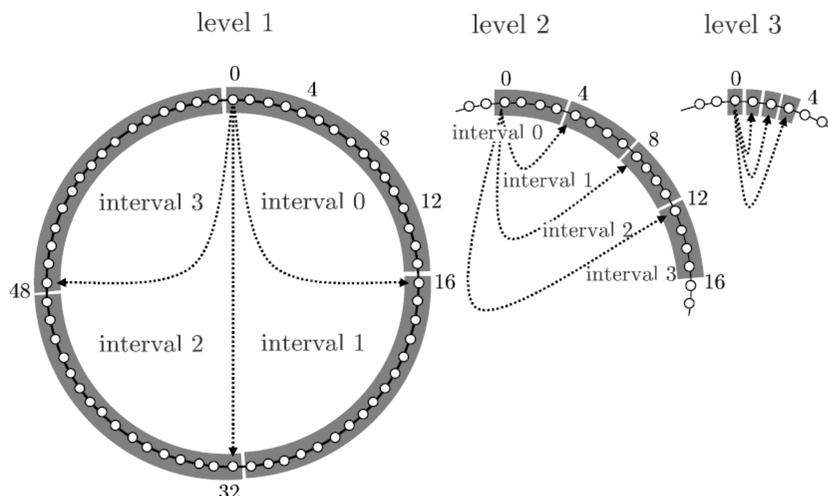


Figure 2.8: Figure taken from [41]. Routing table of node 0, for N = 64 and k = 4. The dotted arrows are the start of the intervals. The dark regions represent the respective intervals. The left most figure shows the intervals on level one. The center figure shows the intervals on level two. The right-most figure shows the intervals on level three.

Low bandwith topology : there is no separate procedure for maintaining routing tables such as Chord (which uses *periodic stabilization*; instead, any out-of-date or erroneous routing entry is eventually corrected on-the-fly thereby, eliminating unnecessary bandwidth consumption. DKS uses two main tecniques called *Correction on Use* and *Correction on Change* which are based on the idea that the routing information is updated whilst the node uses it. In other words, the routing information might be innaccurate if the node does not use it. This way, the messaging system *piggyback* information about the routing entries used in order to correct them in case they were wrong [43].

Efficient Broadcast : efficiently broadcasts a message to all nodes in a ring-based overlay network in $\theta(logN)$ time steps using n overlay messages, where n is the number of nodes in the system. An improvement respect other multicast services in DHTs is its ability to avoid sending redundant messages.

Symmetric replication : is the only one DHT implementation which uses this kind of replication mechanism which makes join and leaves operations very efficient in terms of message complexity.

With respect to the information managed, DKS maintains:

Leaf set : Leaf set contains two lists. The Front list, also called *Successor list*, and the Back list, also called *Predecessor list*. They are used to achieve fault-tolerance in the presence

of failures. Each list contains f pointers to consecutive nodes in clockwise and counter clockwise respectively.

Routing table : The routing table is divided into L rows and K columns. Each entry in the routing table stores the responsible for the interval k of the level l.

Taking into account this information, the routing protocol is similar to the Chord solution. For a given message with a destination *id*, the message is forwarded to the closest node in clockwise direction of the routing table. As each pointer in the routing table divides the space by a k factor, each hop decreases the distance remaining in the identifier space by a factor of k leading to a $\theta(log_k(n))$ complexity in routing messages.

	Chord	Pastry	Tapestry	DKS
System Architecture	Uni-directional and circular space	Plaxton-style mesh	Plaxton-style mesh	Uni-directional and circular space
System Parameters	N-number peers in network	N-number of peers in the network, b-number of bits used for the base (B=2^b) of the chosen identifier	N-number peers in network, B-base of the chosen peer identifier	N-number peers in network, k-number of partitions at each level, f-number fault tolerant parameter
Routing Performance	$\theta(log_2(N))$	$\theta(log_b(N))$	$\theta(log_b(N))$	$\theta(log_k(N))$
Routing State	$log_2(N)$	$log_B(N)$	$Blog_B(N) + Blog_B(N)$	$log_k(N)$
Peers join/leave	$\theta(log_2(N)^2)$	$\theta(log_B(N))$	$\theta(log_B(N))$	$\theta(1)$
Replication Scheme	Primary-backup	Primary-backup	Active replication	Active replication
Replica Placement	Successor-list	Leaf-set	Multiple hash functions	Symmetric replication

Table 2.1: Comparison of various structured p2p overlay networks in terms of its properties and theoretical performance

Chapter 3

Consistency in DHTs

3.1 Introduction

Despite the scalability, availability and fault-tolerance suggested by Distributed Hash Tables, their lack of support for replication of mutable data and for strong consistency models is a problem for becoming a widely accepted model as a viable basis for the future generation of overlay networks.

In this context, *mutable data* should be understood as those data that might be modified during the data life-time. In other words, data which is mutable might be modified instead of only inserted or deleted from the DHT. This new necessity implies a new semantic for the *put* operation of the DHT as well as new mechanisms to guarantee these semantics without losing any property inherent to DHTs.

So far, most DHT-based systems have restricted their focus to immutable data or weak consistency semantics based on cached copies. In such systems, the value of data item can be changed by removing it from the DHT and reinserting it with a changed value. However, this approach will not guarantee that applications will not see older cached copies of data, leading to imprecise consistency semantics. Even if cached copies would be removed, the solution would lead to an important overhead which would derive in an impractical solution.

It is important to note that depending on the application necessity, different approaches to reach consistency are possible, from less consistent semantics to more stricter ones. The most important problem to maintain a certain consistency model in DHTs is its high dynamism and the object replication done to maintain high data availability. Therefore, algorithms to solve this issue should deal with replication as well as with the high dynamisn inherent to DHTs.

Several consistency models have been defined so far [30] to exactly define which kind of consis-

tency is necessary at the application level. Despite that, regarding DHTs, *consistency* is defined as the guarantee to *get* the latest data *put* in the DHT. The definition of when the latest put has been performed is mechanism dependant. Therefore, throughout the document, it will be referred indeferently *retrieving latest data* as *consistent data* or viceversa.

Unlike last efforts with DHTs in sharing a common API[33], consistent data in DHTs is achieved by modifying the algorithm of the *put* and *get* operations in order to guarantee consistency. So there is no common concepts to survey on. Thus, a summary of different approaches taken to achieve consistency is presented in Section 3.3. Despite that, a first classification of different mechanisms to achieve consistency in DHTs is presented in Section 3.2.

3.2 Consistency mechanisms

Despite there is no common mechanism to achieve consistency in DHTs developed in this research area, a first classification can be done depending on the scope of the consistency mechanism. In other words, consistency mechanisms could be divided depending on what element of the DHT should be consistent: *key consistency* or *data consistency*.

3.2.1 Key Consistency

It is defined as, at any given time, no more than one peer claims to be the responsible for a given identifier. These sort of mechanisms deals with the dynamicity and *churn*[1] in DHTs as long as it could lead to incorrect or out-of-date routing tables.

This seems to be a rather modest goal, but in practice is often violated. The most probably reason for that is the stabilization protocol used in most DHTs to correct routing tables as long as each join, leave or failure event lead to a change in the routing table. If these events are not detected correctly, different nodes might have incorrect views of its neighbours (incorrect routing tables). As an example, in a PlanetLab deployment of OpenDHT [44], for the most measured period, around 5% of the keys have multiple responsibles and at certain times the fraction of inconsitent keys spikes to a much higher value (around 40%), presumably due to increased churn.

So the challenge is to keep only one responsible for any given identifier at any time in the lifetime of the overlay network taking into account the dynamicity (nodes joins and leaves) of this peer-to-peer network.

[1] *Churn* in DHTs is defined as the constant leaving and joining of nodes concurrently.

3.2.2 Data Consistency

It is defined as at any given time, a *get* operation returns the last data inserted in the DHT with a *put* operation. So they solve the issue of returning the most current data instead of stalled data taking into account that messages might be lost or replicas might fail.

They deal with replicated data and maintaining data up-to-date even in the presence of dynamic replica set membership and different replica states.

Maintatining replicated data among several nodes in a distributed system taking into account consistency is not an easy issue to deal with as long as there is a really important trade-off between eficiency and escalability.

As it will be shown later, they usually rely on the Paxos algorithm [45] to achieve consensus in order to return the most up-to-date data value in each query or using some kind of ordering making use of timestamping to order the *put* and *get* events [46].

Thus, the challenge is to provide the data related to the latest write operation of an item regardless when this write was performed and regardless the number of failures (in terms of failed nodes or messages lost).

3.3 Survey and comparison

As previously outlined , no common mechanism to achieve data consistency has been developed in the DHT area. In this section it will be presented a survey of differents mechanisms to provide consistency in DHTs by means of the two mechanisms presented previously. Table 3.1 shows a comparison of the forementioned mechanisms.

3.3.1 Atomic Ring Maintenance

This technique stands for the ability to handle multiple joining and leaving of nodes, such that all next successor and predecessor routes still form a valid ring after any number of nodes joined or left concurrently.

DKS holds as the only one DHT which guarantees *key consistency* making use of this technique. In fact, this tecnique was introduced by the developers of DKS as a generic technique applicable for a wide range of existing DHTs.

The aim of the *atomic ring maintenance* is to ensure correctness of lookups in the presence of

nodes joining and leaving. In essence, they serialize interfering joins and leaves, i.e. they are done sequentially, rather than concurrently, to avoid inconsistencies.

The main idea behind this is to modify atomically predecessor and successor pointers to take note of the new joined or left node. Therefore, a *locking* technique is introduced. Basically, each node holds a *lock* which has to be acquired in order to update predecessor or successor pointers. This way, a leaving node should acquire its own lock and the locks of its successor and predecessor. After acquiring all of them it can perform the corresponding updates to maintain the ring consistent. By the other way, a joining node should acquire the lock of the node which performs its insertion and the lock of its future predecessor. The complete algorithms and safety conditions can be found in [41].

3.3.2 Token based root authorization

This technique [47] is based on designating *authorized roots* in such a way that there is never more than one authorized root for a particular key. Therefore, its objetive is to achieve *key consistency* by allowing only one owner for the authorization token. The algorithms are based on DHTs based on ring structure such as Chord or Pastry.

The main idea is to have a single logical bootstrapping node which is the responsible to initiate the authorization algorithm. This node is stable in the sense that it is considered by the algorithm as a persitent service (always available).

This node begins as a responsible of the entire identifier space, therefore is designated as the authorized root for every key (owns each of the tokens). Every time a node joins, this bootstrapping node is responsible to transfer the tokens belonging to the range that the joining node is now responsible for. This algorithm is repeated to adapt to changes in the DHT. Nevertheless this approach is impractical due to the overhead in the bootstrapping node.

One refinement to the previous algorithm is to delegate the transfer of tokens to the owner of those tokens. In other words, a joining node which will be the new responsible for a certain range of keys will contact the current responsible of this interval instead of contacting the bootstrapping node. Despite the improvement reducing the load of the bootstrapping node, this new approach needs to make a round of authorizations to avoid non-availability of authorized nodes due to failures. Therefore, every certain period the bootstrapping node initiates a round of the authorization phase broadcasting a message to all the nodes conforming the ring. This way, failed nodes will release its tokens and they will be assigned to the new responsibles.

The concret algorithms can be found in the cited article but they are quite complex and have hard

constraints. Specifically, they have hard timing contraints assuming that clocks of different nodes advance at exactly the same speed in order to demonstrate their algorithm's safety property. This constraint have been demonstrated as impractical when dealing with distributed systems [46].

3.3.3 Etna: consensus over DHT

Etna [48] is an algorithm for atomic reads and writes of replicated mutable data stored in DHTs. Therefore, its aim is to maintain a consistent view of data items in the face of dynamic environments such as nodes joining and leaving.

Etna is developed on top of Chord to organize a set of cooperating nodes in an overlay network. It is important to note that Etna does not rely on Chord to consistently identify which are the set of nodes which are the closest successors on a object key. In other words, Etna assumes that the set of nodes returned by Chord might be inconsistent so the queries to that nodes might return stalled data.

Etna is based on a primary backup scheme to replicate objects in which the primary replica is the current responsible for an object key and the $f-1$ backup replicas are its closest $f-1$ successors (where f is the replica factor). As explained in Section 2.3.2, the replica set is dynamic in the sense that nodes can join or leave the replica set at arbitrary times. Thus, Etna tries to determine a total-ordering of different group configurations during the life-time of the system. Etna serializes all read and write operations to an item through the primary backup in order to achieve a total order on the operations.

The idea behind Etna is to distinguish two kinds of situations: under normal operation and during a reconfiguration:

Normal operation mode : normal operation is defined as the situa tion where the replica set of nodes is stable. In other words, while the replica set have the same replica nodes. Assuming that the read and write protocol are similar it will be explained only the read protocol: a node p wants to perform a query of the key k. p contacts the current responsible for the k which is r_0. r_0 asks every member of the replica set r_f for the value related to k. If r_0 collects more than $\frac{f-1}{2}$ positive acks, it returns its own stored copy of the object as a quorum has been reached.

Reconfiguration mode : a stable replica set is called a configuration. A configuration has the members of the replica set and the values associated to that replica set. When the replica set membership changes (actually, each time a node joins or leaves the overlay network some replica sets must be changed) a new configuration must be reassembled in order to maintain

consistent results. Etna uses the *agreement protocol* of the Paxos [45] distributed consensus protocol to decide the next configuration. It guarantees that only one value is chosen at most and thus, the replica group will reach a consistent view of the values they are responsible for.

Despite the guarantee provided by Etna, it has a high performance cost in normal operation and during a reconfiguration. During normal operation, $2(f + 1)$ messages will be sent for a single read or write procedure. Moreover, each time a node joins or leaves the network, at least f group memberships change (thus f reconfiguration procedures should be preformed) as, by definition of the replica set membership, each node belongs to the successor lists of its f predecessors. This situation drives to a high message complexity consisting of two and a half round (due to the Paxos algorithm). Every round consists of $2(f - 1)$ messages and so a total message complexity of $5(f - 1)$ messages. Therefore, the joining and leaving procedures of the DHT becomes a high cost algorithm in Etna. Another solution based on Paxos is [49], although it uses Pastry as its underlying overlay network.

3.3.4 Atlas P2P Architecture (APPA)

APPA [50] is a complete architecture for developing complete distributed applications in a fully p2p way. Its main purpose is to hide the underlying p2p infrastructure (namely structured or unstructured overlay networks) to the application. This way, one of its layers is the P2P networks which provides network independence with services that are common to different P2P networks. Two of these services play an important role as much as data consistency is concerned:

Key based storage and retrieval (KSR) : stores and retrieves an item in the p2p network, i.e through hashing over all peers in DHT networks or using super-peers in unstructured overlays. To maintain high available data, it uses the *identifier replication* scheme based on *multiple hash functions* as explained in Section 2.3.2.

Key based timestamping (KTS) : generates monotonically increasing timestamps which are used for ordering the events occurred in the P2P system. Each responsible for a given key maintains a local counter of the last timestamp generated for that key.

So the KSR makes use of the KTS to order update events and therefore it retrieves the latest stored data. In essence, the KSR service provides two basic functionalities. The *insert* functionality stores the corresponding data to an associated identifier. It makes use of the KTS to generate a new timestamp. Then, for each peer which stores a replica of the identifier, it stores the data with the given timestamp. Once the replica receives the item, it compares the stored timestamp

and the received timestamp. If the inserted item is newer, the older one is replaced. The *retrieve* functionality queries for the most up-to-date data stored, or in other words, the data with the highest associated timestamp. First, the node willing to retrieve an item asks the KTS for the latest timestamp generated. Thereafter, it asks each peer holding a replica for the item. If the timestamp of the retrieved element is equal to the one returned by KTS, it means that it is a current replica and is returned as the ouptut of the query. Otherwise, if no replica is up-to-date it returns the item with the highest timestamp.

The main advantage of this proposal is its inherent extensibility as no properties about the underlying overlay network are assumed. By the other way, it has a high overhead related when maintaining up-to-date data due to the queries done to the KTS each time a node performs a lookup for an item. Basically, the cost in messages for a retrieve operation is $2(f + 1)$ assuming each item is replicated f times (two messages for retrieving the latest timestamp from the KTS and two messages for each replica to retrieve its latest data).

	Consistency Focus	Performance		
		Read	**Write**	**Maintenance**
Atomic Ring Maintenance	key	-	-	$\theta(1)$
Token based authorization	key	-	-	$\theta(log(N))$
Etna	data	$2(f + 1)$	$2(f + 1)$	$5(f - 1)$
Atlas	data	$2(f + 1)$	$2(f + 1)$	no mechanism

Table 3.1: Comparison of DHT Consistency mechanisms. They are compared in a qualitative way according the properties introduced in previous sections and in a quantitative way according the number of messages necessary. f is the replication factor of the system.

Chapter 4

Mutual Exclusion in DHTs

4.1 Introduction

One of the fundamental primitives to implement more generic systems and applications on top of DHTs is *mutual exclusion*.

Distributed Mutual Exclusion is a problem that manages the access to a single, indivisible shared resource by at most one process at any time in a distributed environment. In the case of DHTs, the shared resource is each data item stored in DHT and the process is each node in the DHT willing to access the data stored under a given key. These terms will be used indiferently along this chapter.

Although distributed mutual exclusion is an extensiviely studied area [51][52], this chapter focuses on dynamic peer-to-peer systems and, more concretely, on structured overlay networks such as DHTs. As it is a very well known problem, it only will be shown their desired properties on Section 4.2, a simple cathegorization of different proposed distributed mutual exclusion algorithms on Section 4.3 and finally, in Section 4.4, a survey of the only two specific algorithms for mutual exclusion, at our knowledge, developed on top of structured overlay networks.

4.2 Mutual Exclusion properties

In this section it is presented a list of properties and conditions which any mutual exclusion algorithm should provide in order to demonstrate its correctness. Although the conditions used are quite general and could be applied to other scenarios, they will be related to the mutual exclusion problem.

Safety : i.e *nothing bad ever happens* is a necessary condition and is defined in terms of the *mutual exclusion confition*. It asserts that at most one process may execute in the critical section at a time.

Liveness : i.e *something good eventually happens* is defined in terms of the *deadlock freedom condition*. It asserts that a process requesting to enter into a critical section is eventually granted it, as long as any process executing in the critical section eventually leaves it. In other words, no process will be denied to enter its critical section forever (no-starvation).

Fairness : It is a refinement of the liveness property which states that each process has the a similar chance to entry its critical zone to a certain degree. Depending on the degree of flexibility of the algorithm several levels of fairness could be achieved. From more stronger conditions such as *FIFO* in which each grant to enter the critial section is delivered in the same order as the requests were made) to less restrictive such as *the linear wait* in which no process will enter its critical section twice while another process is waiting.

4.3 Classification of Distributed Mutual Exclusion algorithms

Mutual exclusion can be seen as a sort of consistency mechanisms as long as enable nodes to coordinate their activities in order to prevent incoherent behaviour of their operations. If a collection of processes share a resource or collection of resources, often mutual exclusion is required to prevent interference and ensure consistency when accessing those resources. This is the so called *critical section* problem.

In [53], the author argues that despite lots of distributed algorithms have been proposed, only few of them were very innovative, proposing new ideas or new algorithmic techniques. Therefore, two simple ideas drove the construction of any existing algorithm:

Permission-based algorithms : when a process wants to enter into the critical section, it performs a request to the responsible set of granting the access to the critical section for them to give the permission to enter. Then , it waits until these permissions have arrived. If a process is not interested by the critical section it sends back its permission as soon as it receives the requests. If it is interested, a priority has to be established between the two conflicting requests.

The group of processes through which the node is willing to enter the critical section may have arbitrarily configuration. In that sense, each algorithm propose its own configuration depending on the concrete necessities of the system deployed based on the algorithm.

The safety property is ensured by obtaining a sufficient number of permissions of the processes belonging to the responsible set and the liveness property is ensured by totally ordering the requests, usually associating a timestamp to each one.

Token based algorithms : as, by definition, only one process at a time can enter the crticial section, the right to enter is materialized by a special object which is unique in the whole system, namely *a token*.

The safety property is ensured as long as the token is unique in the system. To ensure the liveness property the algorithm must manage the movement of this token from one node to another in order each request to acces into the critical zone be granted.

To manage such a movement, two basic schemes have been proposed. The *perpetuum mobile* allows the token to travel from one node to another to give them the right to enter into the critical section. Arbitrarily network topology is allowed and it only has to ensure that the token reaches every node along the life-time of the system. In the *token asking* scheme, the token does not move itself. Instead, a process willing to enter into the critical section asks for it and waits until the token arrives.

There is a special case when a central coordinator, statically defined, is the responsible to grant the access to processes performing requests to enter into the critical section. This unique permission can be understood as a token managed by this coordinator.

4.4 Survey and comparison

In this section it will be presented a survey of differents mechanisms to provide mutual exclusion over structured overlay networks. Each algorithm is breifly explained and surveyed in terms of the system model (requirements from the underlying overlay infrastructure), the requirements for solving the mutual exclusion and the simple classification presented in previous section.

Moreover, a performance comparison (see Table 4.1) is made in order to clarify the complexity of providing mutual exclusion in such dynamic environment. The performance comparison is made in terms of *message complexity* entering and leaving a critical section, *delay* in terms of message before entering a critial section and its related problems.

4.4.1 Sigma algorithm

The algorithm presented in [54] solves mutual exclusion problem in structured overlay networks where the set of processes may be large, dynamically changing and where processes may crash.

They consider a dynamic structured overlay network where each node might act as a client (which performs the request to enter into the critical section) or a server (which helps to coordinate the client accesses to the critical sections). Nodes can join at any time and servers may crash. They consider that after a server crash, it suffers a complete *memory loss* and restarts itself from its initial state. Thus, its main purpose is to address the mutual exclusion in the pressence of process crashes and memory losses.

They map each resource likely to be acquired in mutual exclusion to an identifier in the identifier space of a structured overlay network. Each resource has its own set of replicas responsible to grant the mutual exclusion.

Basically, when a client wants to enter into a critical section, it sends a *request* message to all replicas. Moreover, each replica maintains a queue with the received requests. Each request has an order id which is its possition in the queue. The replica replies to a request message with a *response* message containing its possition in the queue.

Therefore, the client will be granted exclusive access to the resource if and only if has a majority of responses granting the first position in the queue. The main problem here is the scenario in which the resource is under high contention because each replica will receive the request in different order. Thus, it might be difficult for the client to get a quorum granting its exclusive access as the first in the queue.

To overcome this problem, once the client receives every response and has no access to the resource, it sends a *yield* message. This message is received by every replica and the effect is removing the request from the queue and reinserting it. The function of the yield message is to reshuffle the queue, so it offers a chance to the replicas to build a consistent view and grant exclusive access to a single request.

The Sigma algorithms does not ensure 100% correctnes due to failure. So they rely on the fact that the safety condition is violated with a very low probability. The ensure liveness due to the use of failure detectors and a lease for the clients in order to avoid a client holding a critical section for ever. The algorithm is developed in order to provide a FIFO policy granting the mutual exclusion. Even so, the different order of the messages arrived at replicas and the reshufling of the request queue make difficult to assure this fairness policy.

4.4.2 End-to-End and Non End-to-End protocols

In [55] it is proposed two protocols that combine token and permission based approaches to provide efficient and reliable access to shared resources in dynamic p2p systems.

They model their system as a general structured overlay network where the basic entites in the system are called nodes (or peers), and each virtual resource corresponds to a set of nodes (i.e replicas), where the replicas for a resource are always available but their internal states may be randomly reset due to failures and where nodes communicate via messages across unreliable channels. Its main purpose is to distribute evenly the burden of controlling access to the critical section reducing the overall message overhead.

Both protocols share similarities with the Sigma protocol in terms of the messages sent across the network. Despite that, the treatment of those messages are totally different due to the leverage to some extent of the end-to-end argument exposed in [56], that's why their names.

End-to-End Protocol : the idea is to maintain the queue of requests to enter the critical section at the node that is currently in this critical section instead of at the replicas. It makes use of a quorum set which is defined as the path from the node holding the mutual exclusion to each of the nodes conforming the replica set. Thus, there are as many quorum sets as replica nodes for a resource. If another node wants exclusive access to a resource, it sends a *request* message to each one of the replica nodes. The algorithm makes the assumption that this request will be held by any of the quorum nodes before reaching the resource so the number of messages routed to the shared resources are diminished. Once the request is held by any of the quorum nodes, they forward the message to the node currently in the critical zone which enqueue the request. To acquire the resource in mutual exclusion it has to receive a majority of *response* messages from the replica set or a *token* message from the current holder of the resource in mutual exclusion. This token message carries the list of queued requests at the the time of releasing the resource.

Non End-to-End Protocol : the idea is to maintain a partial queue of requests at all the nodes in the quorum set rather than a complete queue at the critical section owner. This way, the *request* messages are sent to the replicas and handled by any of the quorum sets which enqueue the message itself instead of forwarding it to the current node in the critical section. Nevertheless, once a node exits the critical section, the quorum sets have to consolidate the different queues of each quorum node in order to decide which node will be the next to access into the critical zone.

They do not provide any guarantee in terms of the properties defined in previous sections. Moreover, this solution suffers the same problem of Sigma protocol due to the reshuffling of queues during periods of high contention.

	Classification	Performance	Advantages	Drawbacks
Sigma	Permission based	2k	No mechanism necessary after failures	No FIFO policy, low performance under contention
End-to-End	Permission based	2k	FIFO policy	Low performance under contention
Non End-to-End	Permission based	2k	No single point of failure	No FIFO policy, mechanism needed to reconstruct the waiting queue after failure

Table 4.1: Comparison of Mutual Exclusion over DHTs: k is the number of replicas which manage the access to the shared resource. The number 2 is due to the necessity of a reply, thus the messages sent are k requests and k grants. Thus, performance is measured in terms of messages needed

Part II

The Currency Management System

Throughout this part, we will present the decisions taken to develop our system based upon previously surveyed work. Thus, Chapter 5 presents an overview of the system as well as definitions in terms of the taxonomies presented previously. The fundamental requirements based upon decisions taken within Grid4All are introduced in Chapter 6. Details on the design and implementation of the system is explained in Chapter 8, followed by tests and experiments to characterize it in Chapter 9. Finally, Chapter 10 exposes the project plan and economical evaluation and 11 presents future work and conclusions summarizing each of the contributions made in this master thesis

Chapter 5

General Overview

This chapter defines the currency management used within Grid4All in terms of the concepts explained Chapter 1. It is a simple introduction to the idea on top of which the CMS (and therefore the prototype which is going to be implemented) is designed and developed. They can be seen as general ideas and first decisions taken within Grid4All to reduce the wide range of currency system implementations presented previously.

Although most of the decisions have been taken in the context of the Grid4All European project, the prototype might be applied in any scenario where a distributed banking service is necessary to manage user accounts in order to manage user resource provision and consumption.

Regarding Grid4All, there will be a **unique virtual currency** (at first called *g-currency*) which will serve as a medium of exchange between all trading agents (buyers and sellers). In the context of Grid4All where there will be a common global market in a single instance, we consider as a good choice implementing a single universal currency managed by the currency system of the Grid4All instance. This assumption simplifies currency management and does not imply any loss of functionality. Moreover, it improves the applicability of the currency overcoming the problem of trying to find some other agent with the same currency with which trade.

It should be noted that every Grid4All instance (with its own components) will have its own currency and therefore it will be impossible to transfer funds directly from one currency of one Grid4All instance to another instance. This should be done by means of exchanging to real money in order to buy g-currency in other instances. A unique currency will simplify as well as increase the number of users willing to trade with this currency.

For simplicity, efficiency and for security reasons it would be better to implement the currency as an **account balance based** system. It means that there will be a trusted entity which manages user

accounts and every transaction should use this trusted service in order to carry out the transaction. So there will be a central service in every Grid4All market place responsible for minting and selling specific g-currency for the market place. The alternative of deploying a token-based system was discarded due to the overhead introduced to assure the authenticity of each token (which implies, generally, checking the correctness of a digital signature).

User accounts will be kept in a central banking service. The word central should be understood as administratively centralized. That is, this banking service is a trusted service managed by the Grid4All infrastructure. Despite there is a central logical service which provide g-currency withdraw and deposit operations (currency management), maybe this service should be distributed upon some nodes. The first approach taken for Grid4All will be to organize this service in a DHT (Distributed Hash Table) in a way that each node is responsible for a bunch of accounts. This way, we could achieve load balancing between nodes and make this service fault-tolerant, reliable, dependable, etc. All nodes forming the central service should be considered trusted. So we can classify g-currency storage as a **remote distributed storage**.

This banking service will be the responsible to store a log with the different transactions in order to solve disputes between traders. The resolution of disputes might not be done automatically and might need a third entity capable to solve them.

As long as we consider a banking service which manages user accounts within Grid4All, we can classify the transaction protocols which will be used as **on-line payment protocols**. This way we can assure a certain degree of confidence at the time of transaction checking the account balance before carrying out the transaction. So we can identify users that try to cheat before the transaction is accepted.

Finally, the Grid4All banking service will provide a general enough API in order to allow different kind of transaction orders as for example pay before, during or after use.

Chapter 6

System requirements analysis and specification

Throughout this chapter, we present the first stage for the development of the system: the requirement analysis and its specification. In other words, we present the requirements which the final version of the system resultant from this master thesis must acomplish as well as a first specification of the functionalities provided.

As we have presented in Section , the *Currency Management System (CMS)* will deploy a distributed banking service where users are able to transfer virtual currency from its own account to one another. The reader should note that the users of the CMS are the clients and providers which trade in the market infraestructure of Grid4All. As long as all market related operations in Grid4All are done in the scope of VOs, (*Virtual Organization*) both client VO and provider VO are actually the users of the CMS.

Section 6.1 introduces both user and system requirements as well as outlines the requirements from the infraestructure which will be used for the development of the prototype. Section 6.2 exposes a simple specification of the actors involved in the uses cases (the latter specified in an API way).

6.1 System requirements

We have divided this section in user and system requirements. We consider user requirements are those which describes the system from the point of view of the user. On the other hand, system requirements are those in terms of the system itself ranging from functional to non-functional

requirements.

6.1.1 User requirements

Defining basic user requirements will help us to define the systems requirements which will serve as a guide for the architectural and design decisions. So the Grid4All's banking service:

[1] **Must be high available**. It is an essential service for market related operations. Therefore, the service must be operative every time a user wants to carry out a transaction.

[2] **Must provide basic banking mechanisms**. It is important to maintain the interfaces of the banking service as simple as possible for the sake of ease of use.

- Basic account management: open, close and query accounts.
- Basic transactions: transfer funds and reserve funds.

[3] **Must support different kind of payment methods**. Different acquisition of Grid goods might require different payment methods as explained in Section 1.6. Thus, the system should provide different payment protocols to cover each kind of Grid good transaction.

[4] **Must provide a certain degree of security to users**. It is important that a banking service would be secure enough to avoid misbehaviour of both clients and providers. More concretely:

a user SHOULD be able to:

[4.1] have an identity inside the system that identifies itself uniquely.

[4.2] demonstrate that a transaction has been finished correctly.

[4.3] reclaim any unsatisfactory transaction. The definitive solution to the dispute will be solved by an entity trusted by both traders. An unsatisfactory transaction arises when a supplier does not receive a payment for a service provided to a client or when a client receive an invoice of a payment done for a service it has not asked for.

a user SHOULD NOT be able to:

[4.4] change the balance (currency units) stored in any account arbitrarily.

[4.5] initiate whatever banking transaction on behalf of another user if it has not the correct rights to carry out it.

Optionally, the user would ask for other requirements not fulfilled by this master thesis as long as final decisions are to be taken during the next year of *Grid4All* European Project.

[5] **Must be accessed by means of standard mechanisms**. As long as the Grid is required to use standard mechanisms to stablish communications between different components, the banking service should provide standard communication interfaces to enable users to use this service by means of standard mechanisms such as WSRF or future standards.

6.1.2 System requirements

Once outlined user requirements for the banking service, we describe more precisely the requirements of the CMS banking service upon previously introduced user requirements. We have divided system requirements in *functional* and *non-functional* requirements. Functional requirements are associated with specific functions, tasks or behaviours the system must support. Non-functional requirements are constraints on various attributes of these functions or tasks. Usually contraints about efficiency or robustness not directly related on the functionality provided by the system.

6.1.2.1 Functional requirements

We consider functional requirements as statements of services the system should provide and how the system should behave in particular situations. Most of the concepts used here were introduced in Chapter 1. At the end of each requirements, there is a list between brackets of the user requirements related:

[6] **Must provide a simple but extensible API to work with {2,3}**. Extracting the basic functionalities of a banking service facilitates the implementation of further protocols on top of these basic functionalities, providing a layer architecture which makes a system more extensible. Moreover, it makes easy for the user of the banking service to interoperate with it.

Developing a simple API fulfill the first requirement while enabling the second requirement to be fulfilled. In other words, despite that simple methods will be provided (such as *pay before, during* and *after use*) at first, the interface should be extensible enough to support other protocols introduced later in the development cycle (such as *subscription to an account*).

[7] **Must support logging of transactions during a reclamation period {4}**. The system must log every transaction made in each account ensuring the persistance of that log during a certain period of time during which it is possible for users ask for resolving a dispute or conflict.

[8] **Must manage consistently concurrent transactions against the same accounts{2}**. At any time, users should be able to carry out any request to the banking service and it must

reply with a consistent response depending on past transactions. Thus, concurrent requests to the same account must be handled in a seamless isolated way.

[9] **Must protect users against misbehaving of other users {4}**. Once a payment is made, a non-repudiable proof is sent to both client and provider in order to enable future reclamations (such as the invoices in the real world). Moreover, should provide mechanisms for both actors such as:

A *provider* could inquire a user to send its total balance amount by means of a non-repudiable mechanism before accepting a transaction. This protects the provider from a debit client.

Once a payment is made, the provider has a non-repudiable proof (i.e electronic invoice) in order to demonstrate that it has provided a requested service and has been payed for it. If a consumer tries to claim that a payment has been made when it has not indeed, the provider can win the dispute presenting this proof.

Once a payment is made, the client has a non-repudiable proof (i.e elctronic invoice) in order to demonstrate that it has made a payment. If a provider claims that a payment has not been made when it has indeed, the client can win the dispute presenting this proof.

Should provide non-repudiable operations in order to detect or protect users against over-spending, counterfeiting, etc.

6.1.2.2 Non-functional requirements

We consider non-functional requirements as constraints on the services or functions offered by the system such as development process constraints, design constraints as well as functionalities that must be incorporated to the infrastructure in order to fulfill user requirements. Thus:

[10] **Must be *dependable* {1}**. That is, the system must be operative at any time independently of network conditions and host's load. Therefore the design should take into account the replicated nature of fault-tolerant services.

[11] **The system must support ACID[1] transactions{8}**. In order to avoid inconsistent account balances and reply with consistent results to user requests, the system must carry out operations with ACID semantics. Moreover, transactions such as *transferFunds* must modify two accounts or any one at all in an atomic way.

[1] **A**tomicity, **C**onsistency,**I**solation,**D**urability

[12] **The external interface must be interchangeable by future components {5}**. Despite there is no standards defined to access Grid4All infraestructure services, the system must be designed with interoperatibility, extensibility and exchangeability in mind to enable future access methods decisions (in terms of WSRF, WS [57], Fractal [58] or ProActive[59] development paradigms, etc.)

6.1.3 Infrastructure requirements

Relaying in the user and system requirements, we specify what components, mechanisms or capabilities the infraestructure should provide in order to achieve the desired requirements. By infrastructure, we consider the mechanisms not directly related with the CMS components and, therefore, provided by other components external to the CMS. Thus:

[13] **Security Infrastructure**.

Must provide a way to identify each system user uniquely.

Must provide a mechanism to authenticate each message in such a way that the message can not be forged or falsified and can not be changed without noticing it.

Must provide a mechanism to demonstrate that a message has been sent by a concret user without ambiguities. Moreover, other users must be able to bound a message with and identity.

Must provide a mechanism to authenticate and authorize users to carry out operations they have rights to carry out and deny those they have not.

[14] **Storage infrastructure**. Must provide a distributed storage mechanism with replication techniques to achieve a certain degree of fault-tolerance.

Later chapters will stretch out more information regarding the storage infrastructure. Regarding security requirements, they can be fulfilled by means of a *PKI* component which provide mechanisms to authenticate messages such as cyphering or message signature. They will not be treated throughout the rest of the document as long as the introduction of such security enhances will be a matter of future work.

6.2 System specification

6.2.1 Actors

In this section we specify the actors which interoperate with the CMS components. These actors represents different roles inside the system so each user is not bounded to a concrete actor. Instead, each user can act as different actors(i.e. a provider might act as a client when using resources from another provider VO).

As it is shown in Figure 6.1 the different actors involved in the CMS operations are:

Grid4All user . It is the general identity within Grid4All (or whatever scenario the CMS is deployed). This identity contains a uniquely identification which distinguishes each user from each other. For example, each Grid4All user may own a public/private key pair in order to authenticate itself against the rest of users.

- **CMS Admin** . This user is responsible to resolve disputes or conflicts referring user payments and restore the correct balances in case of missbehaving of users or inconsistencies. It is not a regular Grid4All user as long as its credentials allow it to change arbitrarily the balances of users.
- **CMS User** . They are the users which interoperate with the main operations of the CMS. Thus, each CMS User will own an account created in order to do or receive a payment.
  - **Client VO** . It is the user which initiates the CMS transactions. In other words, the client is responsible to carry out a payment for resources consumed inside its VO.
  - **Provider VO** . It is the user which receives credits to fund its account. In other words, the provider is the CMS User which receives payments on behalf of its VO.

CMS System . This actor represents the CMS component of the Grid4All infrastructure. It is involved in every operation as long as is the responsible to manage the account balances. As we will see in later sections, this actor will enclose different roles inside the CMS component as long as the CMS is a distributed infrastructure.

6.2.2 API Specification

The use cases will be explained in terms of the external API which will be accessed by the actors of the system. Each method corresponds to an use case and is presented in terms of its parameters and which responsabilities have each actor. The actor in *italics* is the initiator of the use case.

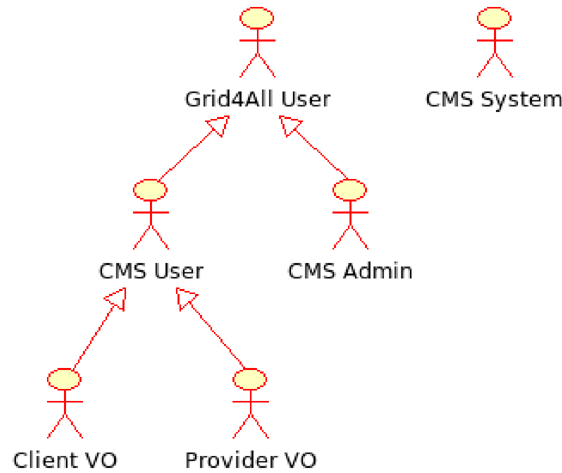


Figure 6.1: Both Client and Provider are Grid4All users. Client will be the responsible to initiate the payment mechanism by transfering funds to its provider

So far, data types used throughout the API specification have not concrete attributes or operations. We introduce them now to clarify the operation signature and the data related with each transaction.

Receipt. Each operation returns a receipt. This receipts will contain information according the result of the operation made. Moreover each receipt will contain security related information which will enable users to present it as a proof of a finished operation.

Account. Each user registered within the CMS, will own a unique account containing its identity, its balance, and the history of operations made by its user.

AccountID. Each Account will be bounded to a unique identifier AccountID. This AccountID might be arbitrarily assigned or be bounded to the identity of the user owning the account.

Credentials. This type represents the rights of a user to carry out certain transactions. These Credentials are bounded to the user identity within the system.

Table 6.1: Banking Service API Specification

openAccount	
Description	Creates a new account and relates it with the given credentials. The user is responsible to present right credentials to the banking service and the system is responsible to bound the credentials to the AccountID assigned. For the sake of simplicity, each user will have only one account associated.
Actors	*CMS User*, CMS System
Input	*Credentials*: unique identification within the system of the user willing to carry out the operation.
Output	*OpenAccountReceipt*: receipt specifying the ID assigned to the user acount.
Exception	*AccountAlreadyCreated*: the account has been previously created for the given user *InvalidCredentials*: the given credentials are not valid within the system

closeAccount	
Description	Closes and deletes the account given its AccountID and the user Credentials. Depending on the system policies, the credits inside the account might be lost or might be refunded to another account. The system is responsible to check if the Credentials are assigned to the given account.
Actors	*CMS User*, CMS System
Input	*Credentials user*: unique identification within the system of the user willing to carry out the operation. *AccountID id*: id of the account to be deleted
Output	*CloseAccountReceipt*: receipt containing the account deleted with the history of transactions.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *InvalidCredentials*: the given credentials are not bounded to the account trying to be deleted.

queryAccount	
Description	Returns the associated account identified by the given AccountID and an identity within the system.
Actors	*Grid4All User*, CMS System
Input	*Credentials*: uniquely identification within the system of the owner of the account *AccountID*: id of the account to be queried
Output	*AccountReceipt*: receipt containing the account queried, including the balance and the history of transactions.
Exception	*AccountNotFound*: the account does not exists within the system *InvalidCredentials*: the given credentials are not bounded to the account trying to be queried.

depositFunds	
Description	Increase the user account associated to the AccountID with the given credits. This method will be only accesed by system administrators to deposit funds due to any reason (i.e. user wins a dispute against a provider).
Actors	*CMS Admin*, CMS System
Input	*Credentials*: credentials of a system administration. *AccountID*: id of the account to be increased. *int amount*: amount of credits to fund the account balance.
Output	*DepositAccountReceipt*: receipt containing the account increased, the reason and the new balance.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *InvalidCredentials*: the given credentials does not correspond to an authorized user.

withdraw	
Description	Decrease the user account associated to the AccountID with the given credits. This method will be only accesed by sysops to withdraw funds due to any reason (i.e. user loses a dispute against a provider).
Actors	*CMS Admin*, CMS System
Input	*Credentials*: credentials of a system administration. *AccountID*: id of the account to be decreased. *int amount*: amount of credits to withdraw from the account.
Output	*WithdrawAccountReceipt*: receipt containing the account decreased, the reason and the new balance.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *InvalidCredentials*: receipt containing the account increased, the reason and the new balance.

transferFunds	
Description	Transfer an amount of creadits (funds) from the source account to the destination account. This operation is atomic in the sense that both accounts are modified or none at all.
Actors	*CMS Client*, CMS System
Input	*AccountID source*: id of the account to be decreased. *AccountID destination*: id of the account to be increased. *int amount*: amount of credits to be transfered. *Credentials*: uniquely identification within the system of the user owning the source account.
Output	*TransferReceipt*: receipt demonstrating that the transfer has been correctly finished.
Exception	*AccountNotFound*: any of the accounts does not exists within the system. *InsufficientFunds*: the source accounts has not sufficient funds to transfer to the destination account. *InvalidCredentials*: the given credentials are not bounded to the source account.

reserveFunds	
Description	Allow the owner of an account to reserve the amount specified for a future payment against the destination account. This operation does not imply a real transfer of funds since the destination account is not modified. This method assures that the reserved funds cannot be spent in another transaction.
Actors	*CMS Client*, CMS System
Input	*AccountID source*: id of the account to reserve funds from. *AccountID destination*: id of the account against which bound the reservation *int amount*: amount of credits to be reserved *Credentials*: uniquely identification within the system of the user owning the source account
Output	*ReserveReceipt*: receipt containing the amount reserved, the source, destination account involved and an identification of the reserve.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *InsufficientFunds*: the source accounts has not sufficient funds to carry out the reservation. *InvalidCredentials*: the given credentials are not bounded to the source account.

commitReservation	
Description	Finishes the reservation of funds transfering the reserved funds specified in the ReserveReceipt to the provider's account specified in the receipt.
Actors	*CMS User*, CMS System
Input	*ReserveReceipt*: receipt containing the necessary information to complete the reservation. *Credentials*: uniquely identification within the system of the user owning the source account
Output	*TransferReceipt*: receipt demonstrating that the transfer has been correctly finished.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *ReservationNotFound*: the receipt specifies a reservation transaction not existing in the source account. *InvalidCredentials*: the given credentials are not bounded to the source account specified in the receipt.

cancelReservation	
Description	Cancels the reservation of funds made before due to some reason. Once the reservation is cancelled the fund are again available for spending.
Actors	*CMS User*, CMS System
Input	*ReserveReceipt*: receipt containing the necessary information to complete the reservation. *Credentials*: uniquely identification within the system of the user owning the source account
Output	*CancelPaymentReceipt*: receipt containing the reservation cancelled.
Exception	*AccountNotFound*: the account does not exists within the system (not cread yet or already deleted) *InvalidCredentials*: the given credentials are not bounded to the account trying to be modified.

6.3 System Model

Defining a system model and its associated failure model is usefull to determine the architecture as well as the design of the components building the system. Relying on this system model, we can identify different problems likely to ocurr and which operations are more frequently executed. So, guaranteeing that the system model is not broken, the specifications and safety properties related to the system are maintained.

We assume a dynamic, cooperating set of node in a totally asynchronous environment. Comunication links may be arbitrarily slow. Nodes can crash (fail-stop), join or leave the system at any time. However, as long as the Currency Management System will be deployed on a relatively stable overlay network, joins or leaves (i.e. due to maintenance) are more frequent than failures due to stopping the node arbitrarily. So our system should provide good performance when peers join and leave the system as well as provide guarantees when peers fail (despite the associated cost).

As we will show in Chapter 9, the probability of braking the safety properties of our system are *negligible*[2]. As we will show, the probability of breaking the safety property of our system decreases exponentially as long as the number of replicas are increased. Therefore, choosing a correct replica factor is important to provide certain guarantees.

[2]Negligible must be understood as the property by which the probability of breaking the correct behaviour of the system is less or equal to the probability of breaking the system model

Chapter 7

System Architecture

Throughout this chapter, we present a general and abstract model of the Grid4All market place architecture and, moreover, we define more concretely the different logical components of the *Payment module* within which the *Currency Management System* is placed.

7.1 Grid4All Market Place Architecture

The document [60] defines the logical architecture of the Grid4All Market Place (a.k.a *GRIMP*) where the generic components that are fundamental to implement market resource allocations systems are identified. This logical view is presented to ensure that the architecture matches the specific requirements that arise from the environment that Grid4All addresses: the allocation of resources for dynamic, ad-hoc virtual organizations.

This architecture shown in Figure 7.1 is composed of different layers, each of which providing mechanisms to other layers in order to fulfill their purposes:

Application Layer : this layer contains the end user applications such as virtul learning applications, simulation applications, etc. Applications are hosted within virtual organizations. In other words, the applications execute within the environment provided by the virtual organizations so as to satisfy the objectives of the VO.

Market Place Services Layer : this layer consists of concrete implementations of market protocols and trading agents within the framework provided by the market infrastructure layers. This layer also consists of market specific applications that may be developed using the functionalities provided by lower layers.

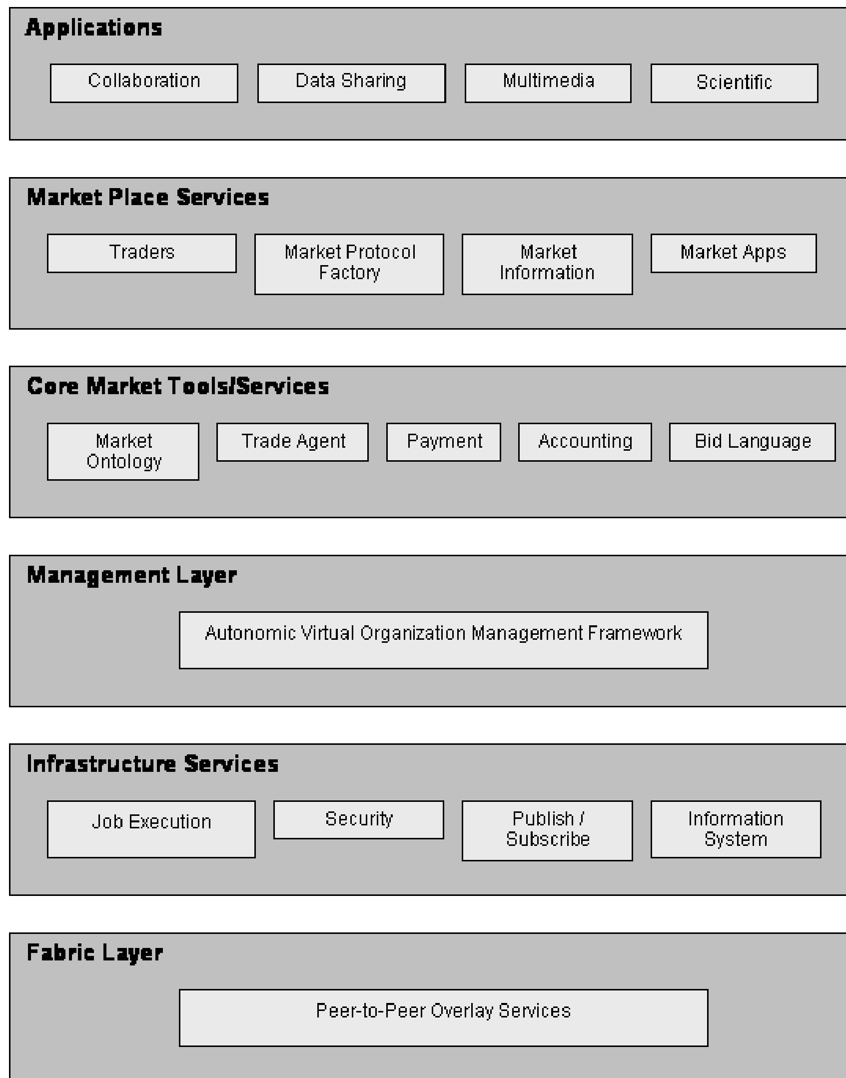


Figure 7.1: Grid4All Market Place (GRIMP) logical architecture.

Market Infrastructure Layer : this layer provides the basic functionalities to implement electronic market places. This layer also provides the framework that facilitates the development of specific market mechanisms and algorithms.

Autonomic VO Management Layer : this layer provides the components based on an autonomic framework to construct self-organizing virtual organizations. Also, it provides capabilities to organize resources used within the VO as well as provide resources to users beloging to the VO on demand.

Infrastructure and services layer : this layer may provide generic infrastructure required to implement large scale loosely-coupled and self organizing distributed systems, althought it is

not specific to the GRIMP.

Fabric Layer : this layer provides connectivity, communication and low level discovery of resources within the platform in order to fulfill the requirements of the upper layers.

The perspective of Grid4All is that of an open market place that provides the tools and services to create *spontaneous* markets. Such markets are initiated by the participants (i.e providers, consumers and also 3rd party mediators) on demand and when they are needed. Such markets are shot-lived and will terminate when its objectives are achieved. This chois is motivated by the fact that on unique persistent market for all Grid4All is infeasible (even if devided by resource/application segments). Secondly, it is also infeasible to thinkof direct bilateral negotiations between providers and consumers as the only alternative. Participants, much like eBay, should be allowed to create market sessions encapsulating suitable rules on demand.

7.2 Payment Module logical view

Our system is encapsulated within the Market Place Services Layer, more concretely within the payment module. Therefore, we present a general logical architecture view to illustrate the different components of the payment subsystem and how it is related to other layers. The logical architecture of the payment module is general enough to match different requirements from different scenarios, although we will focus on the Grid4All specific scenario.

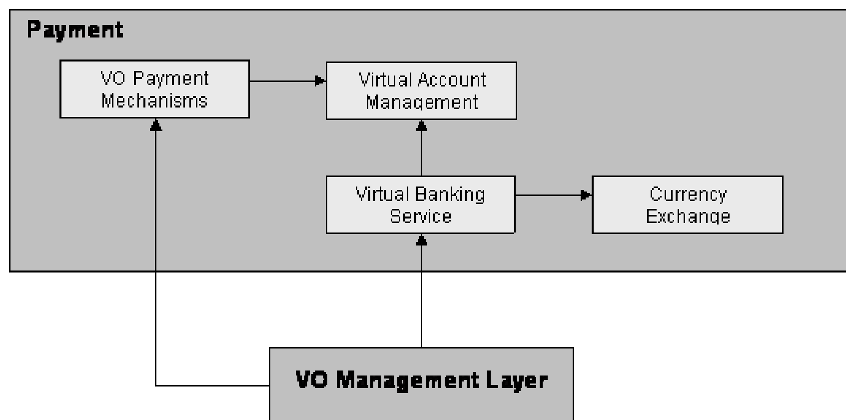


Figure 7.2: Payment module logical architecture.

As presented in Figure 7.2, we have divided the payment module in four basic components:

VO Payment Mechanisms : this component provides different mechanisms to perform the payments due to a Grid4All transaction. These mechanisms may range from pay before use to

subscription mechanisms (See Section 1.6.2). This component the external API to perform those payments.

Virtual Banking Service : this component provides mechanisms to create or delete user accounts and to relate it to a Grid4All User Identity.

Virtual Account Management : this component provides mechanisms to modify user accounts when performing a transaction. Also, it provides certain guarantees (such as ACID properties) when modifying user accounts.

Currency Exchange : this component provides mechanisms to exchange real currency to virtual currency and viceversa. This component may support a wide range of electronic payments such as PayPal or direct Credit Card payments.

The scope of this project is to develope the *Virtual Banking Service* as well as the *Virtual Account Management* as long as these components would be generic enough to be applied in other scenarios. The *VO Payment Mechanisms* as well as the *Currency Exchange* modules are system dependant. In other words, they rely on the specific policies of the institutions deploying the payment subsystem.

Regarding Grid4All, the VO Management Layer is the responsible to initiate the transaction of transferring funds from the account of the VO to the account of the provider. This transfer of funds will be performed by means of the mechanisms provided by the VO payment mechanisms.

7.3 Currency Management System Architecture

Throughout this section, we provide a general view of the CMS architecture. So far, the CMS integrates the Virtual Banking Service and the Virtual Account Management components in a distributed fashion, which deals to a high available fault-tolerant internet-scale service.

7.3.1 Deployment View

Regarding where the CMS will be deployed, we assume a cooperating set of nodes building the whole Market Place. This Market Place might be based on the Autonomic Virtual Organization Management Framework which provides communication primitives based on the Fabric Layer (DHT based Peer-to-Peer Overlay). One of these primitives is the creation of different groups on top of the former overlay. What it basically does is creating another overlay on top of the former one by means of which the nodes can communicate independently of other groups.

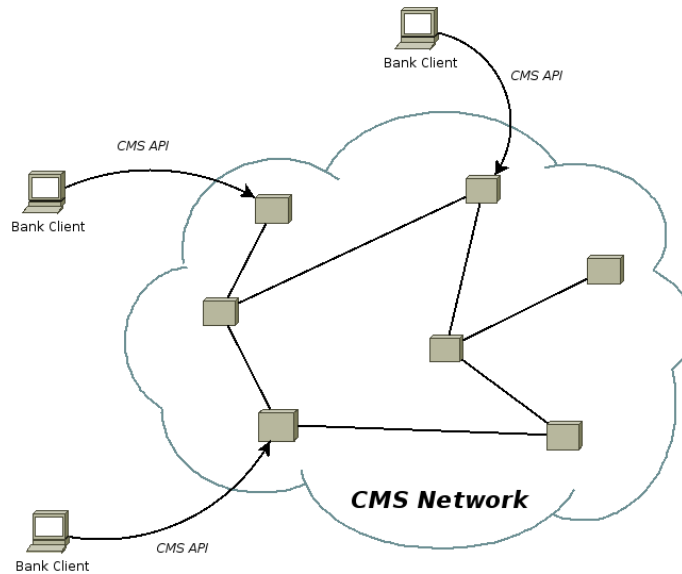


Figure 7.3: Currency Management System deployment view. The CMS Network is constructed upon some nodes each of which is running the prototype of the CMS. Clients access the CMS functionalities through any of the nodes through the external API.

This way, the CMS might be deployed upon a set of nodes of the Market Place. We envisage a distributed high-available fault-tolerant system which is accessed by clients (i.e. Virtual Organizations) through any of the nodes of the CMS Group. This way, the failure of a node does not imply the failure of the entire system but only the failure of a single node. The service may be accessed through another node. Moreover, the possibility to access through any of the nodes implies load balancing of user requests. (See Figure 7.3).

7.3.2 Component View

Once presented the general architecture of the CMS, we introduce the software architecture which will guide the design of the prototype. As we have said, the CMS will be distributed upon some nodes. This way, the basic component of the CMS will be the node which is part of the CMS Group. The architecture of the CMS Node is based on a layered architecture isolating the responsabilities of each one of the layers and providing stronger mechanisms in the upper layers in order to fulfill the requirements of the CMS.

Figure 7.4 shows the layered architecture of the CMS Node. Each layer has its own limited responsabilities abstracting them to the rest of the layers. This architecture enables the system to exchange whatever component without affecting the rest of the tiers.

CMS Gateway Interface Layer : its responsability is to provide a mechanism to export the ex-

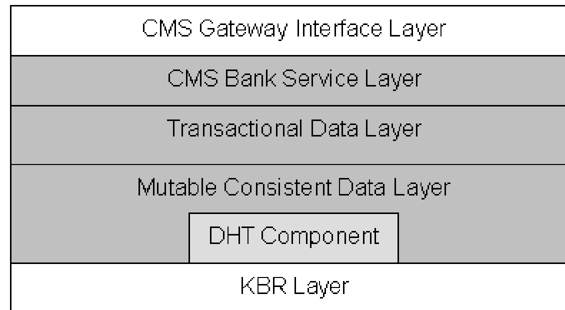


Figure 7.4: Currency Management System Component Architecture. The CMS follows a layered architecture isolating different responsabilities in each layer.

ternal API of the lower layer (Banking Layer) to enable users to access its operations. This layer may be provided by different mechanisms such as WebServices, WSRF [57], Fractal [58] or simply by an internal protocol developed within the system (in the case of Grid4All, it would be developed using the communication primitives provided by the Fabric Layer).

CMS Bank Service Layer : its responsability is to provide operations to perform to account creation and deletion as well as modifications to these accounts when performing, for example, a transfer of funds. It relies on the guarantees supported by the lower layer (Transactional Data Layer) to ensure the ACID properties of its operations.

Transactional Data Layer : its responsability is to provide mechanisms to perform ACID transactions when modifying objects stored in the lower layer. To provide those semanthics, the data will be accessed in mutual exclusion to avoid transaction inconsistencies (as explained in Chapter 4). It relies on the guarantees supported by the lower layer (Mutable Consistent Data Layer) to store and retrieve the objects.

Mutable Consistent Data Layer : its responsability is to provide a slightly modified DHT interface to support the *update* operation as well. Moreover, it is responsible to deliver the most up to date data stored with the semantics explained in Chapter 3. This Layer will be based on the DHT Layer provided by the DKS peer-to-peer middleware as it will be explained in later chapters.

KBR Layer : KBR stands for *Key Based Routing*. As its name suggests its responsability is to provide mechanisms to communicate different nodes based on their key interval responsability (See Chapter 2). We will use the KBR Layer provided by the DKS middleware without any modification. As long as DKS uses a standard KBR API, this layer would be replaced by any other middleware providing this kind of routing mechanisms.

This layered architecture enables us to exchange whatever component in the future. Chapter 8

presents a concrete view on the protocols and design decisions to achieve the responsabilities of each layer.

7.3.3 Roles

The last issue regarding architecture is the roles which can be exerted by each one of the nodes. Based on the ideas presented in Part I we can identify three different roles in our system. These roles may be optional (not every node in the system should exert this role) or mandatory, and will serve as a basis to describe protocols and algorithms in later chatpers:

Gateway Node : also called *Entry Node*. Is the responsible to process the petition of a client by means of the Gateway Interface Layer. It is a simple gateway between the *world* and the nodes *inside* the CMS Network. It is the only one way to access the external API of the CMS. This role is *optional* as long as not every node is required to act as a gateway. Despite that, the more gateways the more distributed is the load of processing client requests.

Responsible Node : As long as the CMS is based on a DHT, each node will be the responsible to perform any operation against an object with an identifier inside its responsability. This role is mandatory as long as the KBR routes the messages to the responsible node of a certain identifier.

Replica Node : To carry out a fault-tolerant system is necessary to do some kind of replication. In our case, what is beeing replicated are the objects stored in the Mutable Consistent Data Layer. The number of replica nodes will depend on the replica factor of the system. This role is mandatory as to produce a fault-tolrerant service is necessary the cooperation of each node to equally balance the replication load.

Chapter 8

Design and Implementation

Throughout this chapter, we introduce the layered architecture within the CMS as well as the algorithms and protocols designed in order to build a distributed banking service.

Figure 8.1 shows a more precise view of the components building each layer. Following a bottom-up introduction of the elements provided by each layer will ease the understanding and explanation of the different solutions presented in upper layers (although the design process was top-down).

Each component belonging to each layer will be presented in subsequent Sections 8.1, 8.2, 8.3 and 8.4. Implementation details of common components or specific features will be introduced in Section 8.5.

8.1 Key Based Routing Layer

This layer provides basic mechanisms to manage a *structured overlay networks*. Its main responsability is to manage the *overlay node* or *peer* and its connections with the rest of its neighbours. More concretely, the main functions implemented in this layer are:

Provide mechanisms to manage network connections : it abstracts the upper layers from the complexity of having to control multiple connections with multiple peers.

Provide mechanisms to manage the overlay : it provides operations to join or leave an overlay as well as to detect failed peers in order to reconstruct its routing table and inform upper layers to fix their state.

Provide basic communication primitives : it provides a simple API to send messages in a syn-

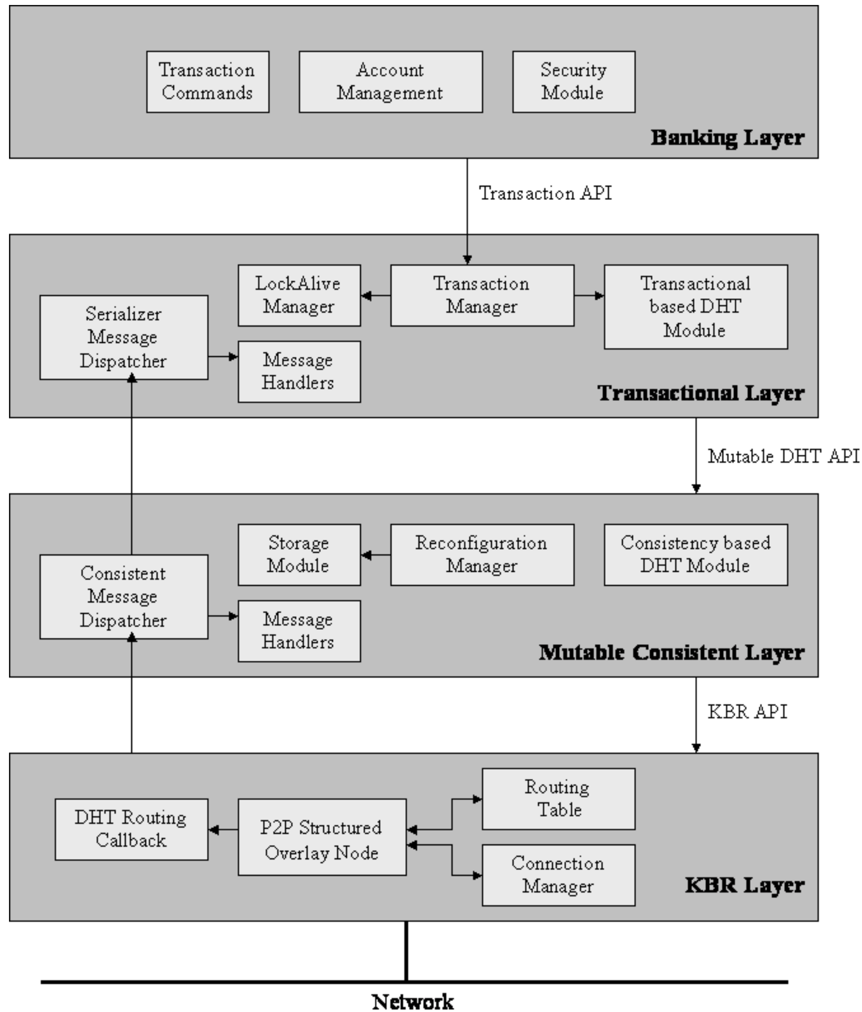


Figure 8.1: Currency Management System layered view.

chronous or asynchronous way as well as broadcast a message to the responsible for a given identifier.

This layer will be the KBR Layer of the DKS peer-to-peer middleware. This decision was made in terms of the properties provided by this middleware, basically the *Atomic Ring Maintenance* and, therefore, its *key consistency property* (See Section 3.3 and Section 2.4.3). Due to this property, harder assumptions can be made to this layer in order to ease the construction of consistency semantics implemented in the upper layer.

8.1.1 Key Based Routing API

Figure 8.2 shows a sketch idea of the KBR API used by the upper layer. Throughout next subsections, each event will be introduced and a complete signature and functionality will be explained.

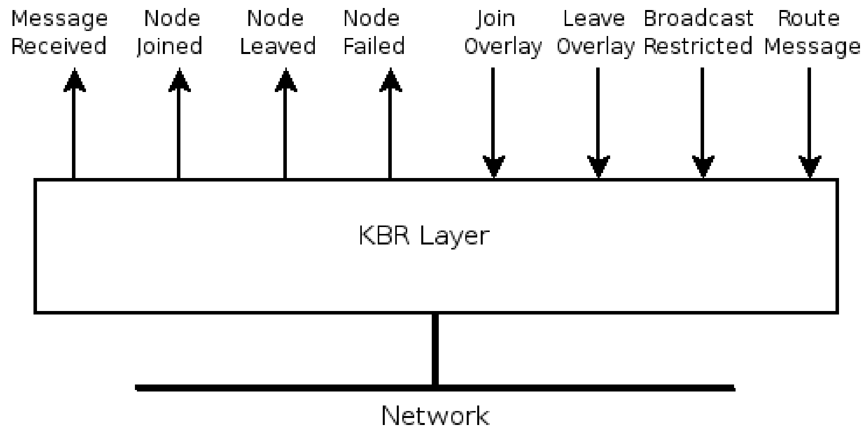


Figure 8.2: KBR Component Interface: lines to the upper layer are notifications of events in which the Mutable Consistent Layer will be interested in order to reconfigure its state. The basic communication primitives used are route a message, and broadcast a message to a set of nodes within the range introduced.

8.1.1.1 Event Notifications

As long as DKS differentiates between synchronous modification of the overlay (namely *join* and *leave*) and asynchronous departures (namely *failure*), three different events are produced to inform the DHT layer. Each event will produce different procedures to reconstruct the DHT layer:

joinCallback : a new node is joining the overlay and it will be its new predecessor.

leaveCallback : an existing node is going to leave the overlay and it was its former predecessor.

failCallback : the overlay node has detected that its predecessor has failed (does not respond to *heartbeat messages*).

8.1.1.2 External invocations

The KBR Layer provides basic communication primitives and mechanism to manage the overlay. Regarding communication primitives we will only explain those used by the upper layer. This way, we decided to build our mechanisms on top of an asynchronous mechanisms as long as not every KBR implementation support synchronous communications. Therefore, the two main operation provided to the upper layer are:

routeAsync : given a number of type *long* representing the identifier within the identifier space and an *object*, this method routes the object to the responsible node for the given identifier. Once the message is delivered to the network, the sender process is able to continue its execution.

broadcastRestricted : given an *interval* (two *long*s representing the start and end of the interval) and an object, sends the object to the nodes which has some reponsability on that interval (may be more than one node). This method is used to recover an interval after a failure as we will see in later sections.

Moreover, this layer provides basic primitives to manage the membership within the overlay. Therefore, the two operation provided are:

join : given an existing node of a certain overlay, this method allows a node to enter the overlay initializing the routing table and neighbour peer connections.

leave : this method allows a node to stop being part of the overlay in a synchronous way by exchanging messages with its neighbours in order to reconstruct their routing tables and maintain the overlay properties provided by the overlay. In the case of DKS, this method ensures that the *lookup consistency* property is held. In other DHTs, this might not be true.

8.2 Mutable Consistent Layer

This layer provides basic mechanisms to deliver the DHT abstractions to the upper layer. In other words, it provides an enhanced common DHT API [33] to fulfill the Transactional Layer requirements and, more generally, the CMS requirements. It is based on the implementation of the DHT layer of the DKS middleware.

As we have said in Chapter 2, most existing DHTs provide good support for immutable data leading to no consistency guarantees. Due to the requirements of the Transactional Layer, our aim is to provide a Mutable [1] DHT abstraction which will not return stale (also *inconsistent* [2]) data regardless of network conditions. Moreover, we modify the identifier representation to enable the possibility of storing more than one value under the same *id* of the identifier space.

As we will show, we modify the semantics of the *put* operation to support *Mutable Data* by replacing it with two new operations such as *createObject* and *updateObject*.

[1] data which can be modified instead of only *put* in the DHT

[2] *Consistency* is defined as the guarantee to *get* the latest data *put* in the DHT

8.2.1 Mutable Consistent DHT API

A first scheme of the API is shown in Figure 8.3. This events will be used by the upper layer (Transactional Layer) in order to build their primitives.Throughout next subsections, each event will be introduced and a complete signature and functionality will be explained.

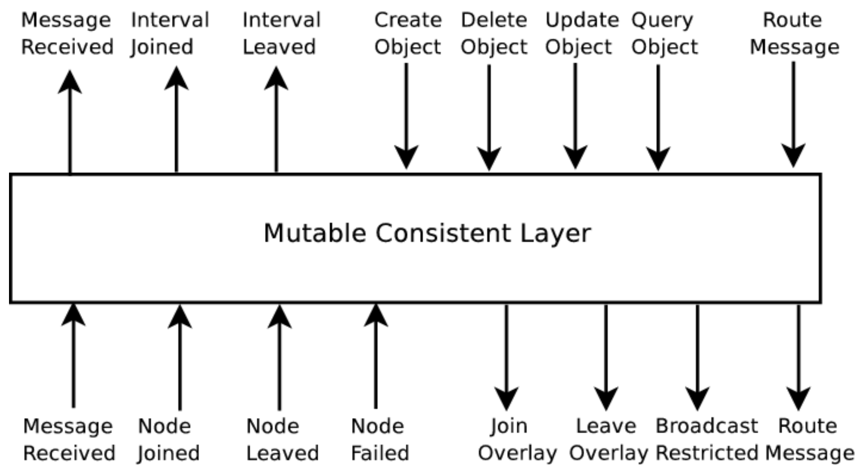


Figure 8.3: Mutable Consistent Component Interface: events sent to the upper layer are intervalJoin, intervalLeave and message Arrived. Moreover, it offers basic enhanced DHT operations in order to support consistent updating of objects.

8.2.1.1 Event Notifications

So far, the KBR Layer provided by DKS notifies the upper layer of the join, leave and failure of nodes events. This way the DHT is able to reconfigure its state with the according mechanisms explained previously.

Nevertheless, at the DHT Layer those events should be masked in order to isolate overlay related events to the upper layers. Instead, we provide to the upper layer the event of joining a new interval (due to leaving or failure of nodes) or leaving a part of the current interval responsability (due to the join of a new node). This way, the upper layer is not aware of which events have been arised but it is able to reconfigure its state by noticing the presence or absence of a new interval to manage. So, the events generated to the upper layer are:

intervalJoin : this event is generated when a node leaves the overlay or has failed. This way, the node receiving this event will be aware of the new interval it is responsible for.

intervalLeave : this event is generated when a new node joins the overlay. This way, the node receiving this event will be aware of the interval it is no responsible to handle from now on.

As we will see, in the Transactional Layer case, the LockAlive Manager component needs to know which objects are beeing held in mutual exclusion to handle the LockAlive messages. If a new node is responsible for a new interval, it must be aware of each new object state in order to initiate a LockAlive mechanism in case the object is locked, or do nothing in case the object is free.

8.2.1.2 External Invocations

This is the first layer implemented by us to support the concrete necessities of the upper layers. In Section 8.5, we will se how the message dispatcher component is implemented and, thus, show how a synchronous communication is possible on top of the basic asynchronous communication provided by DKS. This way, for communication purposes within this layer we provide:

routeAsync : it is, basically, a simple wrapper for the *routeAsync* method provided by the KBR Layer. Despite being a wraper, we check if the message destination is the node itself. If the message is for the node itself, the message is delivered directly to the corresponding message handler as explained in Section 8.5. It is a simple modification made for efficiency purposes.

route : as we will show in a later section, it is a method provided to wait for a response of a message sent. In other words, once the message is delivered to the network, the thread waits till a response for this message is received. Thereafter, the thread is able to continue with its processing taking into account the reception of this response. This primitive allows us to implement simpler methods as long as the the thread is kept waiting while the message is being routed through the network.

As we will explain later, we consider that the *route* method always must receive a response. In order to deal with failures, we have defined a timeout for each one of the messages. This way, if a response for a message is not received after a certain amount of time, the message is resent as long as the previous one might be lost.

Moreover, as we will describe extensively in Section 8.2.3, we provide a simple API representing a DHT. We have enhanced the common API described in [33] to support more stronger semantics such as:

createObject : given an *Identifier* and an *Object*, this methods allows us to bind the object to this identifier within the DHT. If the identifier is in use this methods returns an exception. As we will se, this object will be wrapped within another one to support consistency.

deleteObject : given an *Identifier*, this method allows us to delete the object bounded to this identifier.

updateObject : given an *Identifier* and an *Object*, the object formerly bounded to this identifier is replaced by the new one passed by parameter.

queryObject : given an *Identifier*, this method returns the object bounded to this identifier. If the identifier has no related object, it returns an exception.

8.2.2 Mutable Identifier Space

The basic DHT implementation of DKS enables clients to store more than one object under the same identifier by successively invoking the *put* method. To retrieve one of the objects stored under the same identifier, we must invoke a *get* method and retrieve every item stored previously under a given identifier, leading to bad transmission performance. Moreover, if we store related data (different items which has some point in common) under different identifiers could lead to bad locallity performance due to data being stored in different nodes.

Therefore, we specify a new identifier assignment policy to objects in order to take profit of data locallity and to index each single object stored under the same identifier (avoiding the retrieving of irrelevant data).

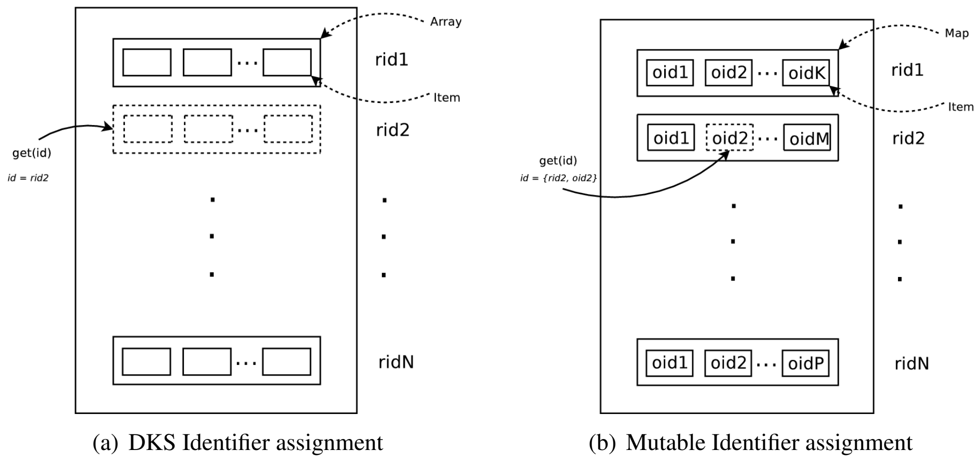
To achieve this purpose, we define a 2-dimensional identifier space where the first coordinate is the current DHT identifier space and the second coordinate is the position within the first coordinate. In other words, we assign each object a *Mutable Identifier* which is constructed by two identifiers:

ResponsibleID : it is the ID of the node responsible to store that item.

ObjectID : it is the ID of the object within the ResponsibleID.

In other words, we propose storing items under the same ResponsibleID within a hashmap for single indexing purposes (indexed by the ObjectID) instead of storing it within an array as DKS do. Figure 8.4 shows the two different approaches taken by DKS and our Mutable Layer respectively.

With this new approach, elements stored inside a single ResponsibleID will remain together despite the joining or leaving of nodes as long as the partition of the identifier space is at the responsibleID level. As we will see, this solution will allow the Transactional Layer to store always under the same node the object stored and its associated state, enabling the efficient retrieving of items.



(a) DKS Identifier assignment (b) Mutable Identifier assignment

Figure 8.4: Figure 8.4(a) shows how DKS manages the get operation taking into account its own identifier assignment. Figure 8.4(b) shows how our Mutable DHT Layer handles the identifier space enabling the retrieving of a single object instead of a set.

8.2.3 Consistency Mechanism

As we have seen, retrieve consistent data in a dynamic asynchronous system is hard to achieve when dealing with replicated data. Our aim is to retrieve always the most up-to-date item stored in the DHT despite the joining, leaving or failure of nodes.

At the end of [41], the author introduces how consistency could be achieved using DKS and its default replication mechanisms, namely *Symmetric Replication*. The idea presented is based on retrieving every item from every replica node and select one of them by achieving consensus (the selected item is the most repeated one). This approach, whilst effective, is less efficient in terms of message cost than retrieving only one item, due to the message overhead introduced. Despite that, this approach is consistent with the parallel *put* operation implemented by DKS.

We have based our solution on the DKS DHT layer by using its symmetric replication technique for efficient join and leave operations and to maintain the *lookup consistency* property. Despite that, the basic operations (namely *put*, *get*) are changed to meet our needs.

Assuming that each node n with identifier i has its predecessor with identifier j, our protocols are based on maintaining a simple invariant which is as follows:

The node n with an interval responsability of $(j, i]$
has the most up-to-date value associated v to each key k
where $j < k \leq i$.

Considering that there are no failures, no joins and no leaves, the invariant holds by serializing every put and get operation through the responsible node for a given identifier. Notice that DKS provides *lookup consistency* which assures that no more than one node will be responsible for a single identifier. This way, each get operation will return always the last put performed.

The challenging issue of delivering consistency semantics is deal with replicated data as well as dynamism. Therefore, we introduce the well known idea of *timestamping* to order different updates which may arrive at replica nodes in different order.

Each object will have an associated timestamp which will be increased monotonically each time an update is performed by the responsible node. This way, a replica will update its own item only if the timestamp associated with the update is greater than the timestamp of the object it maintains. Notice that there is a total order on the set of timestamps generated for the same key. However, there is no total order on the timestamps generated for different keys. It is not a problem for achieving consistency because each item is treated as a different entity.

First, we introduce the different protocols and algorithms to implement the three basic operations of our enhanced DHT abstraction, namely *createObject*, *updateObject* and *queryObject*. For that purpose, based on the replica placement introduced in Section 2.3.2 (more concretely the symmetric replication approach), the replicated identifiers are defined as follows:

$$r(i,x) = (i \ + \ (x-1)\frac{N}{f}) \ mod \ N$$

where i is the identifier, N is the identifier space, f is the replication degree and x is the xth replica within the range [1,f]. This formula matches the Figure 2.5 where different identifiers are replicated along the identifier space of size 16 and replica factor of 4. For the rest of the document we assume that exists a method *associatedIdentifier(i,x)* which given an identifier and a replica number, it returns the replicated identifier for i taking into account the replica number.

Moreover, we assume that each node has an array of hashtables, where each one of the array index (x) stores the replicated items associated with the nodes responsability (taking into account that each identifier is replicated f times). Therefore, every nodes has the items which it is reponsible to store as the primary replica at index *zero* whereas the above formula returns the same id. In the other hand, the rest of array indexes store the *replicated* items associated to the interval responsability.

As we can see in Algorithm 8.1, the createObject message is processed firstly by the responsible node of the identifier associated with the object going to be created. If the object was created, it returns an exception which will be caught by the source of the operation. Otherwise, the object is routed to the replicas responsible for each one of the associated identifier. Once created the *muta-*

Algorithm 8.1: Create Object Algorithm

```
1:  procedure DHT.createObject(Identifier id, Object o)
2:      responsible ← id.responsibleID
3:      response ← route responsible.createObjectHandler(id, o)
4:      return response
5:  end procedure
6:  procedure responsible.createObjectHandler(Identifier id, Object o) from source
7:      if localHashTable[0].contains(id) then
8:          return ObjectAlreadyCreatedException
9:      else
10:         m.id ← id
11:         m.timestamp ← 0
12:         m.object ← o
13:         for all x in 0 ≤ x ≤ f − 1
14:             replica ← associatedIdentifier(id, x)
15:             routeAsync replica.createItemHandler(id, m, x)
16:         end for
17:         return ACK
18:     end if
19: end procedure
20: procedure replica.createItemHandler (Identifier id, MutableObject m, int x) from source
21:     localHashTable[x].put(id, m);
22:     return ACK
23: end procedure
```

bleObject associated to the object, its timestamp is set to zero to start the increasing numbering.

Algorithm 8.2: Update Object Algorithm

```
1: procedure DHT.updateObject (Identifier id, Object o)
2:     responsible ← id.responsibleID
3:     response ← route responsible.updateObjectHandler(id, o)
4:     return response
5: end procedure
6: procedure responsible.updateObjectHandler (Identifier id, Object o) from source
7:     if !localHashTable[0].contains(id) then
8:         return ObjectNotCreatedException
9:     else
10:        m ← localHashTable[0].get(id)
11:        m.object ← o
12:        m.timestamp++
13:        localHashTable[0].put(id, m);
14:        for all x in 1 ≤ x ≤ f − 1
15:            replica ← associatedIdentifier(id, x)
16:            routeAsync replica.updateItemHandler(id, m, x)
17:        end for
18:        return ACK
19:    end if
20: end procedure
21: procedure replica.updateItemHandler (Identifier id, MutableObject m, int x) from source
22:    tmp ← localHashTable[x].get(id)
23:    if tmp.timestamp < m.timestamp then
24:        localHashTable[x].put(id, m);
25:    end if
26: end procedure
```

As shown in Algorithm 8.2, each time an object is updated its associated timestamp is increased by one. This way, when an *updateItem* event is processed by a replica, it will only update the item if the timestamp of the object to be updated is greater than the stored one. Remind that messages may arrive at different order to different replicas and that messages may be lost. This mechanism ensures that each replica stores the latest updated object (to its knowledge) regardless the ordering or losing of messages.

To improve the performance from the entryNode point of view, the responsible node does not wait for the replicas to be updated. Instead, once the object is updated locally by the responsible node and once the messages to the replicas are sent, it sends the response to the entryNode in order to continue with its processing. This way, the consistency will be assured in subsequent queries and

the replicas will be probably updated (this probability will be studied in Chapter 9).

Algorithm 8.3: Query Object Algorithm

```
1:  procedure DHT.queryObject (Identifier id)
2:      responsible ← id.responsibleID
3:      response ← route responsible.queryObjectHandler(id, o)
4:      return response
5:  end procedure
6:  procedure responsible.queryObjectHandler (Identifier id, Object o) from source
7:      if !localHashTable[0].contains(id) then
8:          return ObjectNotCreatedException
9:      else
10:         m ← localHashTable[0].get(id)
11:         return m
12:     end if
13: end procedure
```

Algorithm 8.3 shows the algorithms used to lookup an object stored within the Mutable Consistent Layer. This protocol, by routing the petition to the current responsible node for an identifier, ensures that the latest data will be retrieved. Notice that replicas are not involved with the consequential saving regarding message complexity.

As Figure 8.5 shows, the basic communication protocol of the different roles on which the consistency mechanism relies is as folloes: first, the entryNode sends the corresponding command to the responsibleNode for the identifier. Next, the reponsibleNode sends an update message to each one of the replicas. Finally, the responsibleNode sens a response message to the entryNode to confirm the correct termination of the operation. This is the complete communication protocol for create and update objects. In the case of queries, only the first and third messages are involved.

8.2.4 Maintaining Consistency when dealing with dynamism

So far, we have presented how consistency is achieved by serializing every operation through the responsible node taking into account than no more than one node will be responsible for a single identifier.

The challenge now is to hold the invariant previously introduced in presence of dynamism. Our solution is the same as the DKS proposal when dealing with joins and failures [41] but differs substantially when dealing with failures.

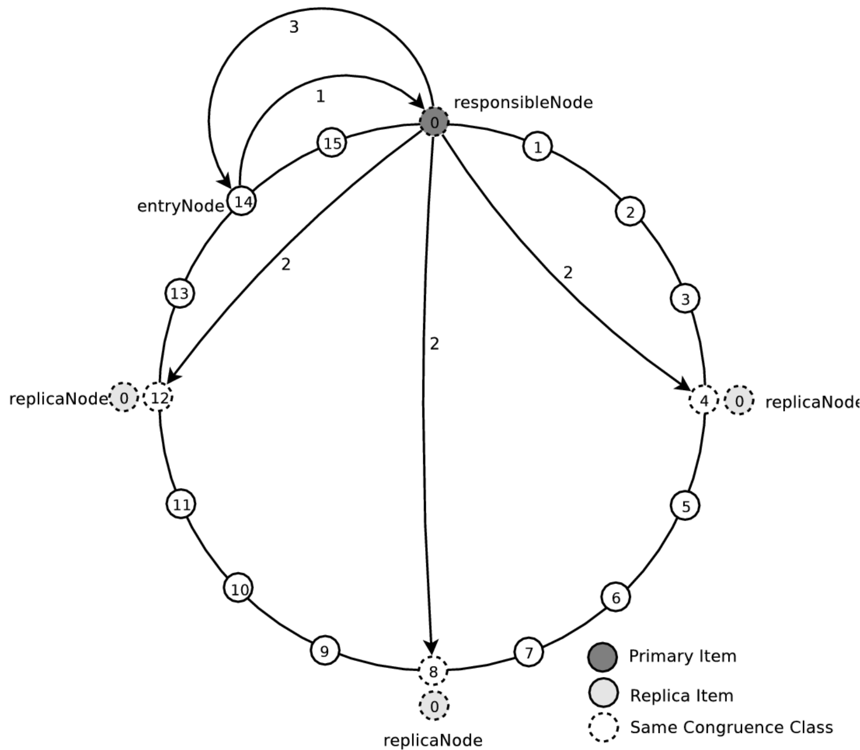


Figure 8.5: Basic Mutable Layer Communication Protocol. Assume a fully populated ring with an identifier space N set to 16 and replication factor f set to 4. Step one: the entryNode sends the corresponding message to the responsibleNode. Step two: the responsibleNode process the message and perform the necessary updates at the replicaNodes. Step three: the responsibleNode sends the response of the operation to the entryNode. In the case of a query operation, step two does not imply updating replica states.

In case of join

The *joining node* retrieves the most up-to-date items from its *successor*. The atomic joining procedure of DKS assures that, while joining, the messages sent to the interval which the new node is responsible for are forwarded to this new node (holding the *lookup consistency* property). Therefore, once retrieved all items, the new node may be able to perform consistent operations against these items. The symmetric replication approach enables the DHT to maintain the replication degree by delegating an interval to the newly joined node (cost of $\theta(1)$) instead of replicating it in the new f predecessors as in neighbourhood replication (cost of $\theta(f)$ assuming a replication factor f).

The invariant previously introduced holds as the new responsible for a certain interval (the joining node) will retrieve the items from the current responsible which has, by definition, the most up-to-date values stored. This way the new responsible will have the most up-to-date values and begin processing new requests consistently.

In case of leave

Similar to the above mechanism, the *leaving node* sends the most up-to-date items to its *successor* as long as it will be the new responsible. As before, the atomic leaving procedure maintains the *lookup consistency* property.

Just like the joining case, the invariant holds as the new responsible for a certain interval (the successor of the leaving node) will retrieve the items from the leaving node which was, by definition, the responsible for the leaving interval, and therefore, had the most up-to-date values.

In case of failure

Failure handling is the main difference regarding items managing between DKS and our solution. Basically, once a node detects that its predecessor has failed (does not respond to *heartbeat messages*), it is responsible for the interval previously held by the failed node. Therefore, the invariant is broken as long as the new responsible does not have the items stored locally. The chanllenge is to reconstruct the interval it is now responsible for in order to satisfy the invariant.

DKS approach is based on retrieving the items within this interval from the first available replicated interval (assuming each interval is replicated f times). This solution may end in retrieving stale data as long as previous messages updating replicated values may be lost due to replica nodes failure. This way, our previous invariant may not hold as long as replica nodes may not have up-to-date values. Moreover, DKS does not introduce the notion of *timestamping* so it cannot distinguish between stale data and current one.

Despite that, we define a mechanism which recovers the most current data at a cost of a more complex mechanism in terms of message and bit complexity. Algorithm 8.4 shows how a failed interval may be recovered. Basically, it sends a *restoreReplicas* message using restricted broadcast mechanism[3] to each one of the replicated intervals in order to recover the values of each one of the intervals.

This message will arrive at each node within the replicated intervals. Each replica node has to send a *recoverItems* message with the replicated items it is responsible to store to the new responsible node. The new responsible node processes each *recoverItems* message by updating the most current recovered item.

Once the interval is received from each one of the replicated intervals, it is able to select the must up-to-date item and store persistenly in its localHashTable. Once every item is recovered, the new responsible node has the must up-to-date value and, hence, the previous invariant is held again. During the *reconfiguration period* (period between the predecessor's

[3] Protocol provided by DKS KBR which broadcasts a message within an interval instead of the whole overlay

failure detection and the final recovery) messages related to objects belonging the failed interval are rejected as long as the node does not fulfill the consistency invariant.

Algorithm 8.4: Interval Reconfiguration Mechanism to deal with node failures

1: **procedure** *DHT*.intervalFailed (Interval failed)
2: **for all** x in $1 \leq x \leq f - 1$
3: replica ← associatedIdentifier(failed.start, x)
4: **restrictedBroadcastAsync** *replica*.restoreReplicasHandler(failed, x)
5: **end for**
6: **end procedure**
7: **procedure** *replica*.restoreReplicasHandler(Interval failed, int replicaIndex) **from** responsible
8: responsability ← getResponsability()
9: intervalToRestore = responsability ∩ failed
10: itemsToRestore = ∅
11: **for all** i in $failed.start < i \leq failed.end$
12: itemsToRestore = itemsToRestore ∪ localHashTable[replicaIndex].get(i)
13: **end for**
14: **routeAsync** *responsible*.recoverItemsHandler(itemsToRestore)
15: **end procedure**
16: **procedure** *responsible*.recoverItemsHandler(Set items) **from** replica
17: **for all** i in *items*
18: recoveredItem ← recoveredInterval.get(item.id)
19: **if** recoveredItem.timestamp < item.timestamp **then**
20: recoveredItem.timestamp ← item.timestamp
21: recoveredItem.object ← item.object
22: **end if**
23: recovered.put(consensusItem)
24: **end for**
25: **if** interval received f times from each replicated interval **then**
26: localHashTable[0] ← localHashTable[0] ∪ recovered
27: **end if**
28: **end procedure**

The main difference between the DKS algorithm and our algorithm is the number of replica intervals queried. DKS queries only one replicated interval and waits for responses. If responses are lost, it retries against the next replicated interval. Our mechanism introduces a parallel interval recovery to every replicated interval at once. If some replicated intervals are not available within some amount of time, the reconfiguration mechanism pick up the most up-to-date value recovered so far instead of expecting the reception of every replicated interval.

8.3 Transactional Layer

So far, we have described how an enhanced DHT is able to provide with very high probability data consistency. This way, through this section, we describe how we use this enhanced DHT to build a distributed mutual exclusion mechanism which enables us to construct transactional semantics on top of it.

As we have seen in Chapter 4, those systems are a step in the right direction to achieve mutual exclusion over structured peer-to-peer networks, but they does not fully utilize the powerfull nature of the DHT domain.

This layer provides simple and lightweight mechanisms to retrieve objects and update them in mutual exclusion. It is important to provide such mechanism to enable us to build transactional semantics and, thus, provide ACID properties to the upper layer. If we does not provide such mechanisms to the banking layer, concurrent modifications to different account may lead to inconsistent account balances by losing updates.

Our idea is, basically, distribute the load of mutual exclusion requests upon the nodes building the DHT network. This way, each node is responsible to allow the access to enter the critical section for a set of objects it is responsible for. According the classification made in Chapter 4 we could describe our algorithm as the special case where there is a central coordinator (*responsible node*) against which nodes ask for the permission before entering the critical section. This unique permission can be understood as a token managed by this coordinator.

8.3.1 Transactional Layer API

8.3.1.1 Event Notification

As long as the main purpose of this layer is to provide a Transactional interface to the banking layer and isolate the upper layer from the complexities of dealing with a dynamic network, we do not notify of any event.

8.3.1.2 External Invocations

For the purpose of our application (the banking layer), we must provide a simple transaction interface as well as a mechanism to create, delete and query objects. The lock managing as well as the update mechanisms are hidden by our Transaction Manager to the appplication layer. To enforce the application to use the transactional mechanism to perform the updates to avoid concurrency

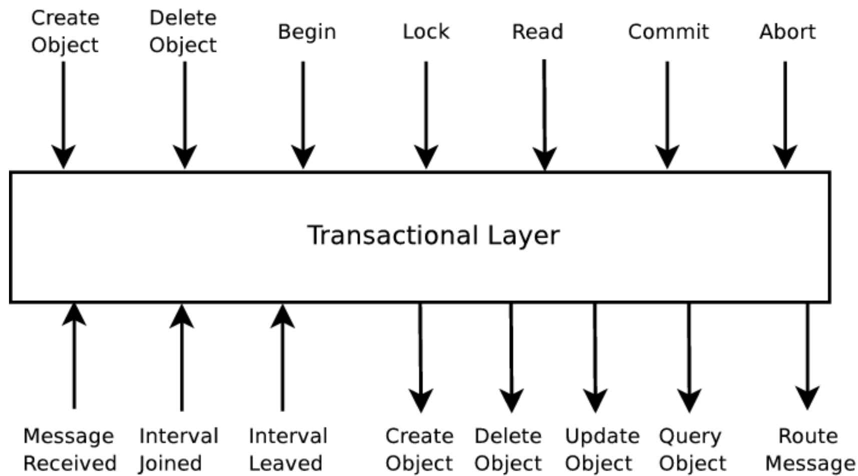


Figure 8.6: Transactional Component Interface: this layer does not provide any information about the underlying overlay network. This way, it abstracts completely the management complexity by providing a powerfull while simple Transaction API.

issues, we do not allow direct updates against the DHT. Complete algorithms and explanations will be presented throughout the next section. Therefore, the external API provided by this layer is built upon the method provided by the Transactional DHT and our Transaction Manager:

Transactional DHT :

- **createObject** : this method allows the application to create an object managed by our transactional layer. It means that, besides the creation of the application object, we must create another one which will represent the state for that object. As we will see, this decision enables us to delegate the complexity of the *transfer state issue* to the Mutable Consistent DHT Layer.
- **deleteObject** : this method allows the application to delete both objects created previously.
- **queryObject** : given an object identifier, it returns the associated object. Notice that it is not necessary to do it in mutual exclusion as long as this query is only for information purposes. If the object is going to be modified, it should be read through the transactional mechanism.

Transaction Manager : there will be a new transaction manager for each one of the transactions willing to be executed at the *entryNode*.

- **lock** : this method adds the given identifier to the list of objects going to be acquired in mutual exclusion.
- **begin** : this method starts the *growing phase* of a transaction. In other words, it starts the acquiring of locks for the objects identified by the identifiers introduced by the *lock*

method. There are different possible policies to acquire the locks. The one we have implemented is the *ordered lock acquirement* which stands for acquiring the locks ordered by their identifiers in ascending order. As we will see, this simple mechanism allows us to avoid dead-locks.

read : as long as the mechanism used within the begin method to acquire locks is the *lockQuery* (see Section 8.3.2), once a lock is grant, the object is returned alongside the object for efficiency purposes. Therefore, this method gives the application the object stored locally when the reception of the object was done.

commit : once the transaction has modified each object accordingly and no exceptions has been thrown, the commit method begin the updating of objects and its corresponding unlocking. This time, the order of updating or unlocking is not relevant. For efficiency purposes, we perform the update and unlock steps within the same message (*commitUnlock*). This procedure is also called *shrinking phase* of a transaction.

abort : if any operation within the transaction has failed or is not possible to perform the modification because of an exception, the transaction has to be aborted, and thus, unlock each of the locked objects without updating them. We consider that a transaction may only fail due to transaction semantics, not for node failures. For example, the *Transfer Funds* transaction may only fail when there are not enough funds in the source account.

8.3.2 Mutual Exclusion Mechanism

To support the above mentioned Transactional API, a mechanism to acquire an object in mutual exclusion is necessary. As mentioned before, current mutual exclusion algorithms for Distributed Hash Tables has a great impact in the number of messages as well as does not provide enough guarantees regarding fairness conditions.

Our aim is to simplify to the maximum the design of our mutual exclusion algorithm as well as to take profit of the DHT capacities. Relaying on the Consistent Mutable DHT Layer, we use the DHT abstraction to store the object willing to be held in mutual exclusion as well as the state of that object. The use of the underlying DHT to store the state for an object enables us to abstract the fault-tolerance problem of dealing with dynamism.

We provide a simple mechanism to acquire the *lock* of an object and *relase* it once the update has been performed. To achieve atomicity we need to serialize the acquisition of those locks. The simplest way is to route the message to acquire the lock to the responsible node for that object. The responsible node will serialize every request to acquire the lock in a FIFO queue. These lock

requests will be served sequentially.

Thus, our algorithms could be classified as a *Centralized Permission-based* approach which stands for a simple and cheap (2-3 messages per request) mechanism providing no starvation and fairness as long as each request is served in FIFO order. Nevertheless, this approach makes the coordinator (the node responsible to manage access grants) a bottleneck. In this sense, we explore in Chapter 9(Evaluation) which load degree the coordinator is able to manage.

8.3.2.1 Stored Objects

For the purpose of a fault-tolerant mutual exclusion algorithm, we use our enhanced *mutable identifier space* to store two different but related objects within the DHT: the object willing to be accessed in mutual exclusion and its unique associated state (Figure 8.7).

TransactionalObject : this object is a wrapper of the object stored in the DHT. We use this wrapper to identify the state associated to this object.

StateObject : this object maintains the queue of requests for the TransactionalObject associated. Moreover, it provides methods to modify and query the state in order to take decisions when processing requests to the associated object. As long as the only one attribute is a queue of requests, the state of the object returned by the *getState()* could be *FREE* (in case the queue is empty) or *LOCKED* (in case the queue is not empty).

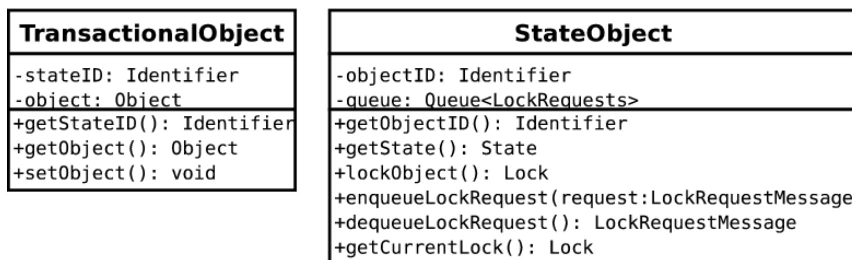


Figure 8.7: Two different objects stored within the Mutable Consisten DHT Layer. The TransactionalObject stores the identifier of its associated state. The StateObject stores the identifier of its associated object and provides some operations to manage it.

The use of two different objects to represent a single upper layer object is due to efficiency purposes. We assume that the associated state for an object is queried and updated more often than the object itself. This way, each time we modify the state for some reason, the object is maintained as it was and no communication overhead is necessary. The operations provided by each object are enough self explanatory.

Despite using two different objects to represent a single upper layer object, we must provide only one identifier to the upper layer. This way, we have decided to relate the identifier of the upper layer object with the TransactionalObject. If we want to access the StateObject, we must query the TransactionalObject first in order to know its associated state identifier.

8.3.2.2 Basic mechanism assuming a stable network

Assume we have a stable peer-to-peer structured overlay of an arbitrary number of nodes and an arbitrary replication degree. Each node will be the responsible node for a set of identifiers.

First of all, we introduce how objects are created and deleted. Basically, the idea is to use the *create* and *delete* operation provided by the Mutable Consistent Layer. As long as each object created through the Transactional Layer is associated to two different objects (the object itself and its associated state), we must create both objects in the DHT using the *create* operation of the underlying DHT. The complexity of managing already created objects is held by the Mutable Layer. In case of an object deletion, we must delete both associated objects. As long as the identifier presented to the upper layer corresponds to the *TransactionalObject*, we must delete the *ObjectState* identified alongside the deleted *TransactionalObject*.

From now on, we explain how we use the underlying DHT infrastructure to modify objects stored in order to acquire such objects in mutual exclusion. Algorithms presented use a *dht* variable which represents a way to perform calls to the underlying Mutable Consistent DHT Layer.

Algorithm 8.5 shows how the lockRequestMessage is processed by the responsible node for an object depending on the state of the object. It begins by sending a *lockObject* message to the responsible node for a given identifier. This identifier is the one associated to the object which is wanted to be accessed in mutual exclusion. The responsible node queries from the underlying DHT both the *TransactionalObject* and the *StateObject*. It is important to notice that both objects will be the most up-to-date objects and that the queries to the DHT are resolved locally as long as the responsible has the most up-to-date objects.

Once obtained both objects, the processing depends on the current state of the TransactionalObject. If it is *FREE*, the request is enqueued to change the state to *LOCKED* and the *Lock* object is created. As long as this *Lock* object is the permission to enter the critical section, it must be unique in the system. For that purpose, we introduce a monotonically increasing timestamp which is increased each time a new lock object is created.

Thereafter, the state is updated in the DHT and the *Lock* is sent back to the source of the request. Upon receiving the *Lock*, the entry node is able to modify the object in mutual exclusion. In the

case of a *LOCKED* object, the request is enqueued in the FIFO queue and the state is updated in the DHT. No responses are sent back to the source of the request as long as the entry node has not acquire the object. As we will see, once the object is unlocked, the next request will be served.

Algorithm 8.5: Lock Object Algorithm

```
1:  procedure Transactional.lockObject(Identifier id)
2:      responsible ← id.responsibleID
3:      lock ← route responsible.lockRequestHandler(id)
4:      return lock
5:  end procedure
6:  procedure responsible.lockRequestHandler(Identifier id) from source
7:      transactionalObject ← dht.queryObject(id)
8:      stateObject ← DHT.queryObject(transactionalObject.getStateID())
9:      if stateObject.getState() == FREE then
10:         stateObject.enqueueLockRequest(lockRequestMessage)
11:         lock ← stateObject.lockObject()
12:         DHT.updateObject(transactionalObject.getStateID(), stateObject)
13:         return lock
14:     else
15:         stateObject.enqueueLockRequest(lockRequestMessage)
16:         DHT.updateObject(transactionalObject.getStateID(), stateObject)
17:     end if
18: end procedure
```

As we can see in Algorithm 8.6, an *unlockObject* message is sent to the responsible node for a given identifier. The responsible node queries both *Transactional* and *State* objects associated to the given identifier. It dequeues the current lock request from the state queue and updates the state object in the DHT. Once updated, an ack is sent to the source of the unlock request to allow the entry node to continue with its processing. Once the current lock request is dequeued, the state might be *FREE* or *LOCKED*. If it is *LOCKED*, the first lock request in the queue will be the current lock request. Therefore, the responsible node locks the object and sends this lock to the source of the current request. This way, the next in the queue will continue its own processing with this object in mutual exclusion.

The algorithm for the *query* operation has the same semantics as the Mutable Consistent Layer one as long as we allow queries without having the object in mutual exclusion. This is due to the fact that its application dependand to know the concret necessities of the application in terms of consistency when reading objects. If the application wants to read the object in mutual exclusion it may perform a lock request before reading it but it is not mandatory.

Algorithm 8.6: Unlock Object Algorithm

```
1:  procedure Transactional.unlockObject(Identifier id)
2:      responsible ← id.responsibleID
3:      response ← route responsible.unlockRequestHandler(id)
4:      return response
5:  end procedure
6:  procedure responsible.unlockRequestHandler(Identifier id) from source
7:      transactionalObject ← DHT.queryObject(id)
8:      stateObject ← DHT.queryObject(transactionalObject.getStateID())
9:      stateObject.dequeueLockRequest()
10:     DHT.updateObject(transactionalObject.getStateID(), stateObject)
11:     return ACK toNode source
12:     if stateObject.getState() = LOCKED then
13:         lock ← stateObject.lockObject()
14:         return lock toNode lock.holder
15:     end if
16: end procedure
```

In the case of the *commit* operation, the only one difference between it and the *update* algorithm of the Mutable Consistent Layer is the lock check. In other words, before updating the object in the DHT, we check that node performing the *commit* operation is the holder of the current lock stored in the StateObject.

With this basic mechanism, the application could acquire an object in mutual exclusion for modifying it without interferences when an stable network (without joins, leaves nor failures) is considered. For efficiency purposes we provide two more operations: *lockQueryObject* and *commitUnlockObject*. The first operation allows the application to lock an object and, as a result, it receives the locked object. In the second one, the application is able to commit a modified object and unlock it in one step without incurring in two rounds of communications.

8.3.2.3 Mechanism to deal with dynamism

Throughout this section we propose a mechanism to deal with the intrinsic dynamism of peer-to-peer overlay networks. We will distinguish between requestor node failure (a node holding a lock stop working) and interval reconfigurations (the interval responsability has changed due to some reasons).

Requestor node failure If a node holding a lock of a certain object fails, it arises an issue regarding the liveness property of our algorithm. In other words, if a node holding a lock fails

Algorithm 8.7: Commit Object Algorithm

```
1:  procedure Transactional.commitObject(Identifier id, Object object, Lock lock)
2:      responsible ← id.responsibleID
3:      response ← route responsible.commitRequestHandler(id, object, lock)
4:      return response
5:  end procedure
6:  procedure responsible.commitRequestHandler(Identifier id, Object object, Lock lock) from
    source
7:      transactionalObject ← DHT.queryObject(id)
8:      stateObject ← DHT.queryObject(transactionalObject.getStateID())
9:      if stateObject.getCurrentLock() = lock then
10:         transactionalObject.setObject(object)
11:         DHT.updateObject(id, transactionalObject)
12:         return ACK
13:     else
14:         return NACK
15:     end if
16: end procedure
```

before performing the unlock, it will be impossible that future lock requests will be served as long as the responsible node will detect that the object is still locked and it will enqueue the request. As long as no unlock requests will be received, the object will remain locked forever.

For that purpose, we introduce the idea of a *LockAlive Manager*. The LockAlive Manager is a component (See Figure 8.1) which is executing at the responsible node. It basically ensures that the entry node holding a lock is still alive after a certain amount of time (namely *lockAlivePeriod*).

This way, each time a node acquires the lock for an object, the LockAlive Manager will send a *LockAliveMessage* to the holder of the lock every *lockAlivePeriod* period. If the holder of the lock does not respond within a *lockAliveTimeout* period, the LockAlive Manager will consider it as failed and will initiate the recover process. The recover process consists of basically unlocking the object following the Algorithm 8.6. This way, the object will be served in mutual exclusion to the next request in the queue. For simplicity purposes, we do not consider *undo* or *rollback* operations without losing any kind of functionality. In other words, if a node commits and object correctly but fails to perform the unlock operation due to a failure, the commited object will persist in the DHT despite the unlock operation has failed.

This situation makes sense as long as if a client performs a commit for an object (without

unlocking), the node wants this modification to be persitent even if it fails. If the application requires atomic commit and unlock it may use the *commitUnlock* operation provided for efficiency purposes.

Interval reconfiguration Interval reconfiguration means the movement of items within the DHT when nodes join, leave or fail. As mentioned in Section 8.2, these three events are masked to isolate the upper layers from the complexity of managing such low level events. This way, our Mutable Consistent Layer offers two differents events which helps the Transactional Layer to reconfigure itself: *intervalJoin* and *intervalLeave*. Moreover, our enhanced DHT ensures that new objects stored will be the most up to date object present in the DHT.

Interval Leave Event : If a new node joins the overlay, its new successor will arise an *intervalLeave* event as long as it will stop being the responsible for a certain interval of identifiers. This event is simple to manage as long as the current responsible node must cancel every LockAlive timer managed by the LockAlive Manager as long as it has no responsability on this interval from now on. The new responsible will handle those LockAlive timers.

Interval Join Event : If a node leaves the overlay or fails, its current successor will receive an *intervalJoin* event as long as it will begin being the responsible for the leaving node's interval. To reconfigure the LockAlive Manager state, it must initiate a LockAlive timer for each new object belonging to this joined interval which state is *LOCKED*. If the state is *FREE* it has nothing to do.

With this two events, the Transactional Layer is able to reconfigure itself and continue managing the requestor failure as if no dynamism where present. Without the presence of these two events and its consequent reconfiguration, dead locks may occur. To show you this case, assume an entry node which is holding a lock for an object and that a lock alive timer is active at the responsible for that object. If the entry node fails and, thereafter, the responsible node fails or leaves before the lockAliveTimeout is expired, the new responsible for that object will not be aware that the lock holder has failed. Therefore, as long as no *unlock* messages will be received for that object, it will always consider this object as locked and subsequent lock requests will be enqueued within the state object.

8.3.2.4 Safety and liveness considerations

Once outlined our algorithms, we demonstrate informally how those maintain both safety and liveness properties without taking into account failures:

Safety property : as we outlined in Section 4.2, the safety property is defined in terms of the *mutual exclusion condition*. It is acomplished by requesting the lock for an object and granting it iff the lock is not held by another process. As long as the responsible node only grants the permission to enter the critical section to one process at a time, there will be only one process able to manipulate the object (fulfilling the mutual exclusion condition).

Liveness property : as we introduced previously, the liveness property is defined in terms of the *deadlock freedom condition* or, more explicitly, no process will be denied to acquire an object in mutual exclusion once a request is perfomed. It is acomplished by storing each request and serving it sequentially once the current holder of the lock releases it. We assume that the critial section processing ends within a finite amount of time and, thus, each subsequent request will be served eventually.

Fairness property : besides the liveness property, we provide a stronger condition, namely *FIFO ordering*. This condition assures that each request will be granted in the same order as they have arrived to the responsible to manage the lock for a certain object. This way, each process will have the same chance to enter the critical section depending on the number of concurrent requests to enter. Moreover, we consider that a process enters the critical section once the request has arrived to the responsible node but the access is not granted until it is the first request in the queue.

Taking into account failures, both properties are satisfied too. The *safety property* is compromised if a responsible node for a certain object fails with an arbitrary object state. This way, the new responsible must retrieve the last modified state in order to continue the execution from the same point. This procedure is also called *state transfer* [35]. We use the underlying DHT to store each state object so the state transfer is done at the DHT level. This way, we rely on the high probability of recovering the most up to date data offered by our enhanced DHT. If an out of date data is retrieved, the behaviour of the algorithm is unexpected as long as two different nodes might acquire the lock. Despite that, the probability of recovering stale data is extremely low.

Regarding the *liveness property*, it is compromised if a current holder of a locked object fails as long as the next requests in the queue will wait forever untill the object is released. We solve this situation by detecting when the holder of a locked object fails by means of the LockAlive Manager component (which checks if the current holder is still alive).

The *fairness property* is maintained taking into account both mechanisms previously mentioned. As long as the queue of requests is stored within the state of the object, a failure of a responsible node does not imply the losing of those requests. Therefore, the FIFO ordering is preserved once the object is recovered. In the case of a lock holder failure, once the LockAlive Manager detects

that it has failed, it will fire the unlock procedure and, therefore, grant the next request in the queue to enter the critical section.

8.3.3 Transactional Mechanism

Once outlined how to achieve an object in mutual exclusion, we introduce how a transactional mechanism providing ACID properties is constructed on top of it. Basically, we offer the application basic operation to deal with transactions in such a way that the complexity of acquiring objects in mutual exclusion is hidden.

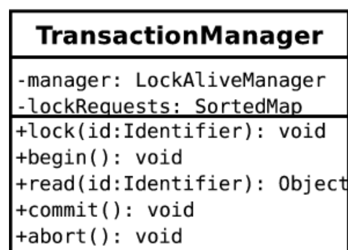


Figure 8.8: TransactionManager Class Diagram

Basically, as shown in Figure 8.8, what we offer are operations to create, delete and lock objects as well as begin, commit and abort transactions. As explained in Section 8.3.1, we offer a simple interface to begin, commit or abort a transaction.

We must take into account that dealing with transactions might produce deadlocks as long as each transaction may acquire shared resources (accounts in our case) in mutual exclusion. There are four necessary conditions for a deadlock to ocurr:

Mutual Exclusion : each object is being managed by a single process or any at all.

Hold and Wait : each process is waiting for acquiring more than one resource in mutual exclusion to execute its critical section.

No preemption : an object only is released if the holder of the lock do it. No other process is able to release an object on behalf of the owner.

Circular Wait : there is a circular chain of two or more process where each one is waiting to acquire an object which has been previously acquired by the next member of the chain.

Avoiding one of those four previous conditions, the system is free of deadlocks. Erasing the *Mutual Exclusion* and *Hold and Wait* are imposible as long as our system needs both. Providing

preemption to our mutual exclusion algorithms would arise more complex mechanisms to manage the lock as long as different locks for the same transaction are managed by different nodes (different responsible nodes for different objects).

This way, the easiest way to avoid deadlocks is avoiding the *circular wait condition*. This is done by applying a global order when trying to acquire a lock in such a way that every process acquire locks in the same order. This order will be the natural order of the *object identifier* (which is basically a number of type *long*).

Therefore, the *begin()* operation acquires locks following the object identifier order. In other words, it begins acquiring the object with the lowest identifier and, once obtained, it acquires subsequent objects following an ascending natural ordering.

Nevertheless, this policy may be system dependand as long as the probability of acquiring conflicting objects may vary. In a system where the probability of acquiring the same object in mutual exclusion is low, this ordering mechanisms would deal to high delays due to waiting for resources to be acquired. In this case it would be better to try to acquire locks in parallel (without any prestablished order) and try to detect when a deadlock has occured. We have decided to implement the order mechanism for simplicity although in future versions it would be possible to extend the implementation to detect when deadlocks occur.

To sum up, using this transaction interface, the application is able to implement operations which need stronger guarantees decoupling the complexity of acquiring and releasing locks on demand and hidding the overlay network management. Regarding the application view, it will only interact with this interface as if the operation was performed locally to the application.

Algorithm 8.8 shows an skeleton of a transaction, based on which every transaction can be implemented.

Algorithm 8.8: General Transaction Skeleton

Require: *objects*: list of ids which the transaction wants to acquire in mx
Require: *manager*: responsible to hide the complexity of managing mx. External Interface.

```
 1: procedure Application.runTransaction()
 2:     for all id in objects
 3:         manager.lock(id) /* Introduce objects to be acquired in mutual exclusion */
 4:     end for
 5:     manager.begin() /* Growing phase */
 6:     object1 ← manager.read(id1) /* Read object with id1 */
 7:     object2 ← manager.read(id2) /* Read object with id2 */
 8:     ...
 9:     /* Modify objects accordingly */
10:     if No Exceptions then
11:         manager.commit() /* Shrinking phase with updates */
12:     else
13:         manager.abort() /* Shrinking phase without updates */
14:     end if
15:     return result
16: end procedure
```

8.4 Banking Layer

Having a simple transaction interface to work with, it is very simple to construct the banking layer as long as the complexity of managing overlay related events is hidden. This layer contains three basic modules: *Transaction Commands, Account Management* and *Security Management*.

8.4.1 Transaction Commands Component

We have decided to build this layer based on the Command Pattern in which objects are used to represent actions. A command object encapsulates an action and its parameters. This way, the *CMS Gateway Layer* could be implemented in any way without concerning which commands are available or, for instance, to apply priority policies on different commands. It is a common way to implement a command to be executed in a different thread. This way, a thread pool containing different threads may execute whatever command as long as each command is independently defined with its own related information.

Figure 8.9 shows different commands implemented in the Banking Layer in order to fulfill the API presented in Table 6.1 (CMS Banking Layer API).

Notice that the *Command* abstract class has two implemented methods to store and retrieve the

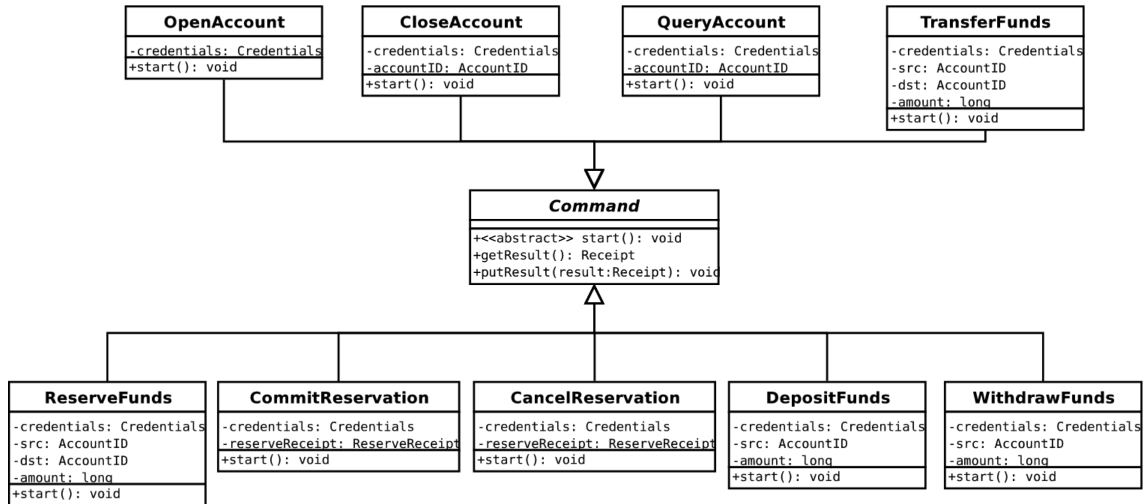


Figure 8.9: CMS Banking Layer Commands

result of transaction. This result will always be of abstract type *Receipt* to ease the handle of different result type.

8.4.2 Account Management Component

The basic entity to manage user accounts are *Account*s. This entity is the one stored within the DHT regarding the banking layer. The entity *Account* is composed, as shown in Figure 8.10, of:

an AccountID : this object allows the system to uniquely identify the Account. This accountID will be used to index the account within the DHT.

a Credential : this interface allows the system to compare the current identity of the account owner and the identity of the user trying to perform the transaction. If the credentials are not correct, the system does not allow to perform the operation and the transaction is cancelled without any modification. As for now, we have implemented a *SimpleCredentials* class which contains a single string. If the strings are not equal the *checkCredentials operation* return false. Envisaging the future, this class will wrap some kind of cryptographic information which enables the system to cryptographically identify the owner of this account.

a Balance : within this object we store the current balance of the user account as well as the balance reserved for future transactions. We provide to simple methods such as *increase* and *descrease* both balances in order to allow transactions to modify them accordingly.

a list of TransactionLog : each TransactionLog will be usefull for two different purposes taking profit of polimorphism:

- maintain a log of every transaction finished correctly within this account and, therefore, serve for the purpose of logging.
- encapsulate information regarding a single transaction and, therefore, implement each kind of transaction separetedly.

This way, when *addTransaction* is executed, the *Account* appends the current transaction beeing processed in the list of transactions (logging) and then execute the transaction to perform the actual operation being in course transparently (modify balance).

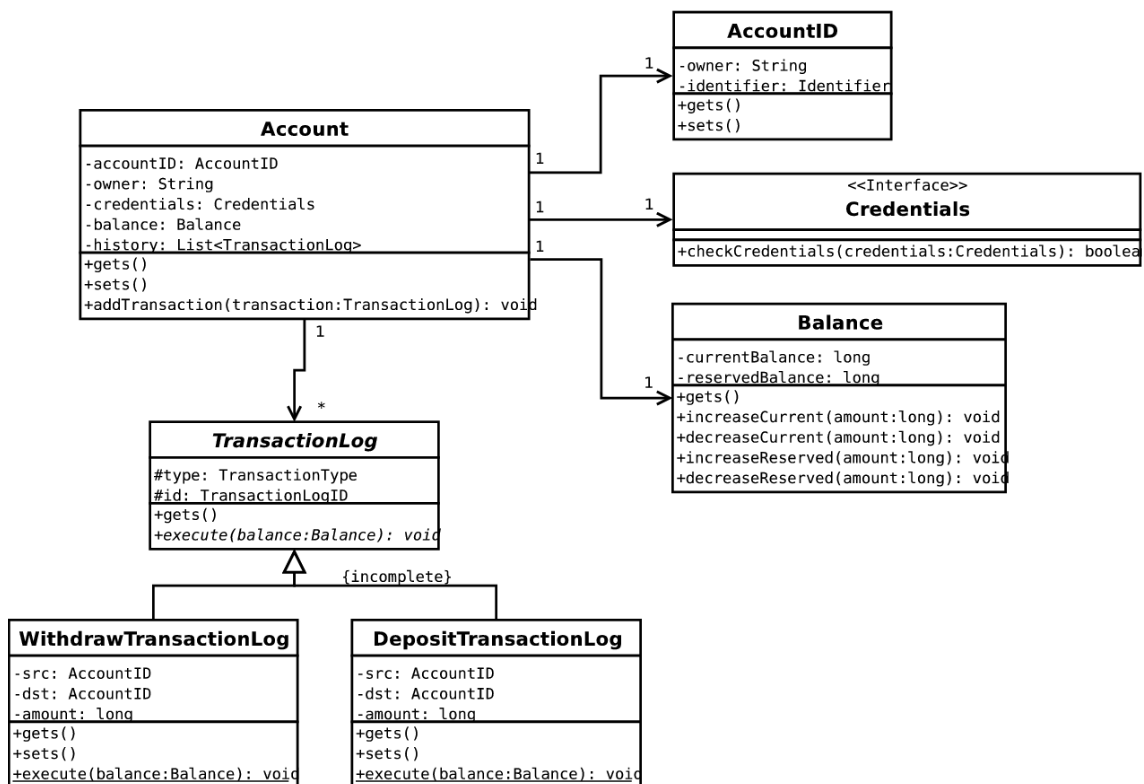


Figure 8.10: Account and Receipts Class Diagram

8.4.3 Security Management Component

So far, Grid4All has not taken any kind of decision regarding security management. We will be discussing it in the near future taking into account the security requeriments presented in Section 6.1.3.

As a summary and to show the reader how the interaction is done between these different components we show the complete sequence diagram of the *transferFunds* transaction assuming that two accounts have been created previously (See Figures 8.11, 8.12 and 8.13). The rest of transactions follow the same scheme changing only the error handling and the account modifications made.

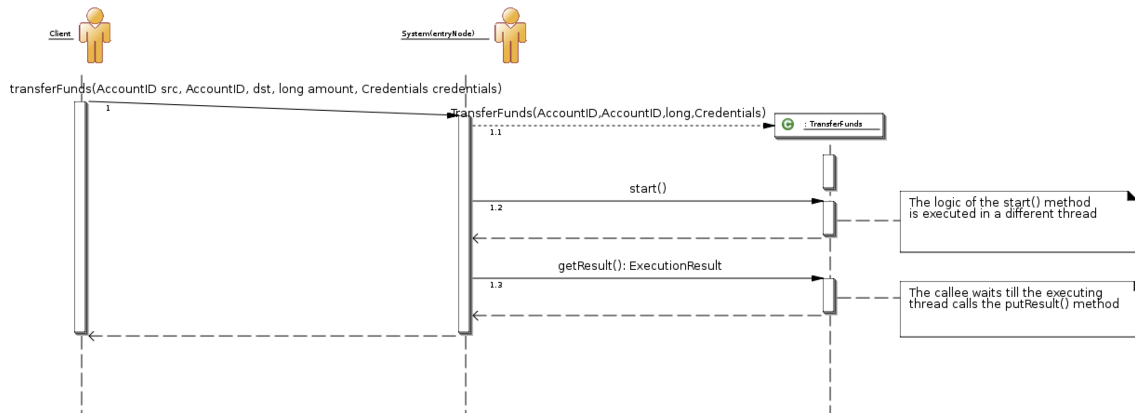


Figure 8.11: Example of CMS API call through an abstract gateway node. The client asks through a previously known mechanism (Web-Service, WSRF, Internal Protocol, etc.) to a concrete API method. It shows how the entry node creates the Command object and execute it in a separate thread.

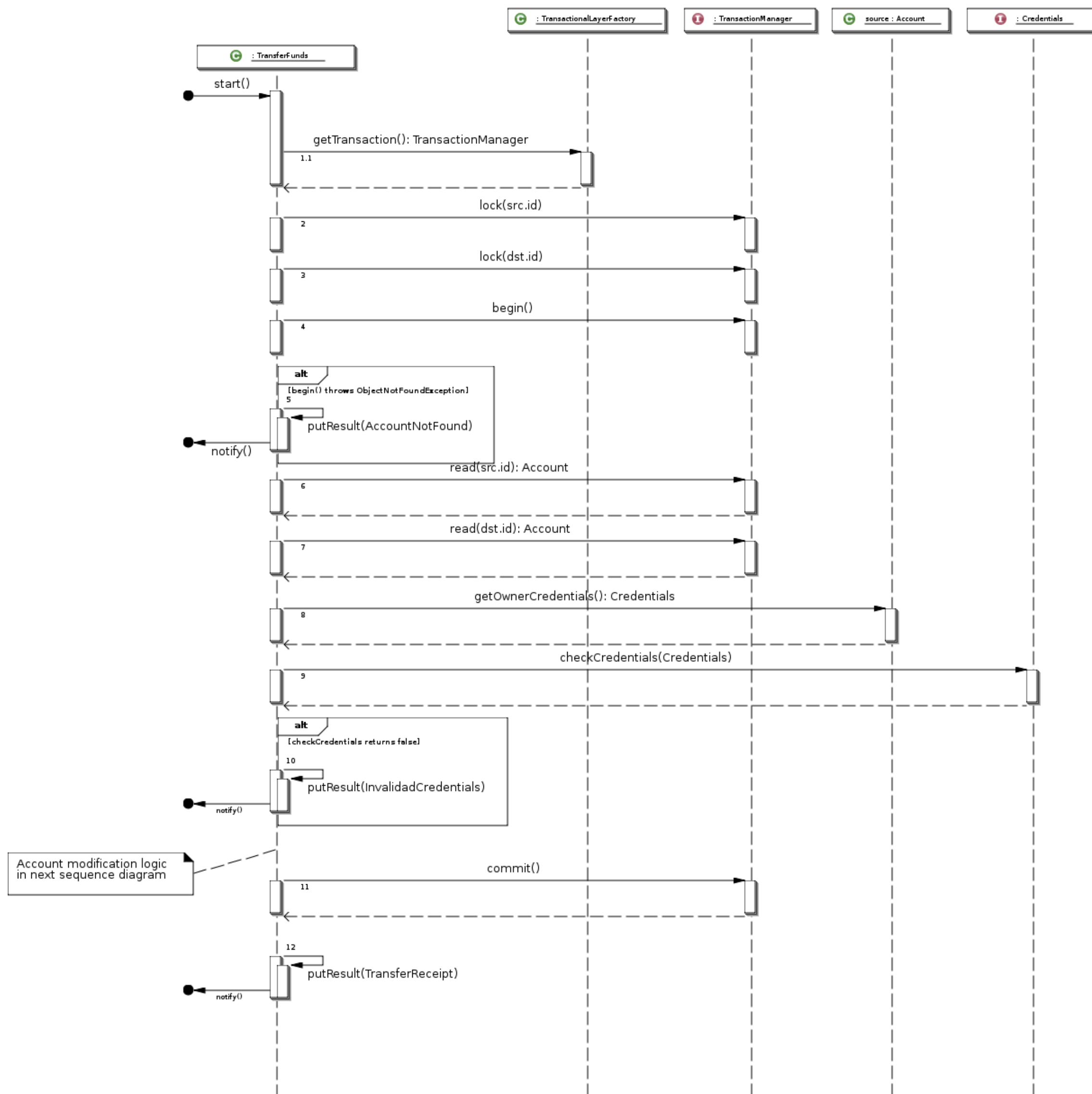


Figure 8.12: Example of a transaction execution following the general transaction algorithm, more concretely the TransferFunds Transaction. The figure shows how the transaction asks for a TransactionManager, which will be the responsible to manage the locking and unlocking of objects within the TransactionalLayer. First of all, it asks to lock the source and destination accounts through their identifiers (specified within its AccountID). Thereafter, the transaction calls begin() which actually begin to requests the locks. Once obtained, the transaction read the locked values (this read is local as long as the lock mechanism retrieves the account as well). Finally, it checks for correct credentials, execute the according modifications and commit the results if no exceptions has been thrown.

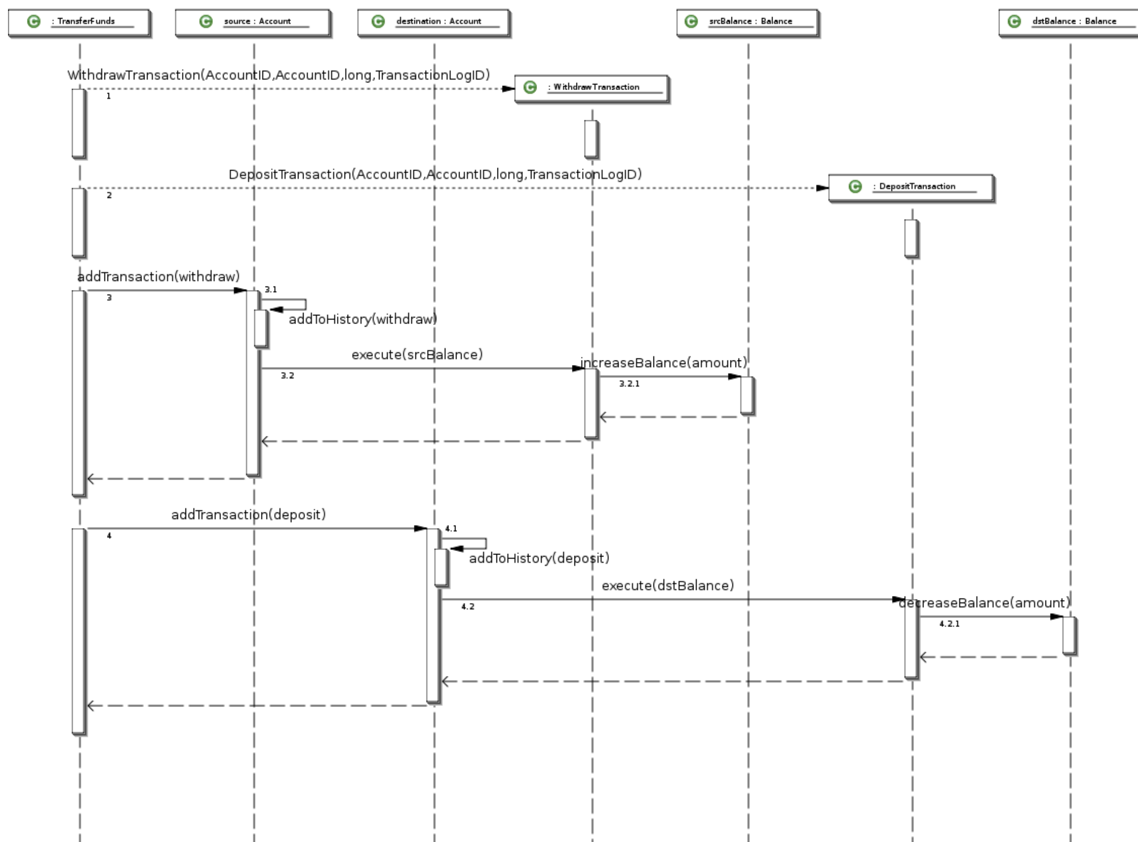


Figure 8.13: Example of the logic of the TransferFunds transaction. It shows how both deposit and withdraw transaction logs are created for both account modifications. Once created, each transaction log is inserted within the account which means: a) add this transaction to the history of transactions and, b) execute the transaction which is to increase or decrease the balance of the account depending if the transaction is deposit or withdraw respectively.

8.5 Implementation details

Throughout this section we introduce common components to every layer in more detail showing the reader some sequence diagrams which will help to understand how we achieve certain properties explained in previous sections.

8.5.1 Message Dispatching Component

As long as our aim was to develope a middleware independently (to a certain degree) of the underlying KBR Layer, we have developed our own message dispatching mechanism. The introduction of this component allows us to develope a synchronous communication without concerning how the asynchronous mechanism is provided by the lower layer.

Figure 8.14 shows the class diagram of the message dispatcher component. Each message dispatcher has a thread pool responsible to execute handlers associated to each message type. As long as the thread pool has a limited size of threads executing actions, we must encapsulate the necessary information to execute the handler associated with a message in a class named *MessageDispatcherJob*. As long as it implements the Runnable interface, the ThreadWorker is able to execute this job. This job basically contains a *MethodInvoker* (also message handler) which encapsulates the method to be executed and the object defining such method. This class invokes the method within an object dynamically by means of the *reflect package* which helps to instantiate and call methods given their string definitions (complete class definition in the case of instantiation, complete signature in case of calling methods).

We have implemented a basic message dispatcher which sequence interactions are shown in Figure 8.15. Despite this basic implementation, different layer might need different approaches to deliver messages to their associaciated message handlers. Notice that each message dispatcher might have a callback message dispatcher used to deliver messages which are not meant to be handled within this layer.

For this purpose, we have extended the basic implementation to accomodate each layer to its own specific requirements:

MessageDispatcherReconfiguration : (See Figure 8.16). Messages arrived to this layer while the node is reconfiguring its state (retrieving items from the replicas to select the most up-to-date) will lead to inconsistent results as long as the node would receive a message to modify an object which is actually being recovered. This way, if the destination of the message belongs to an interval which is under reconfiguration, the message is discarded.

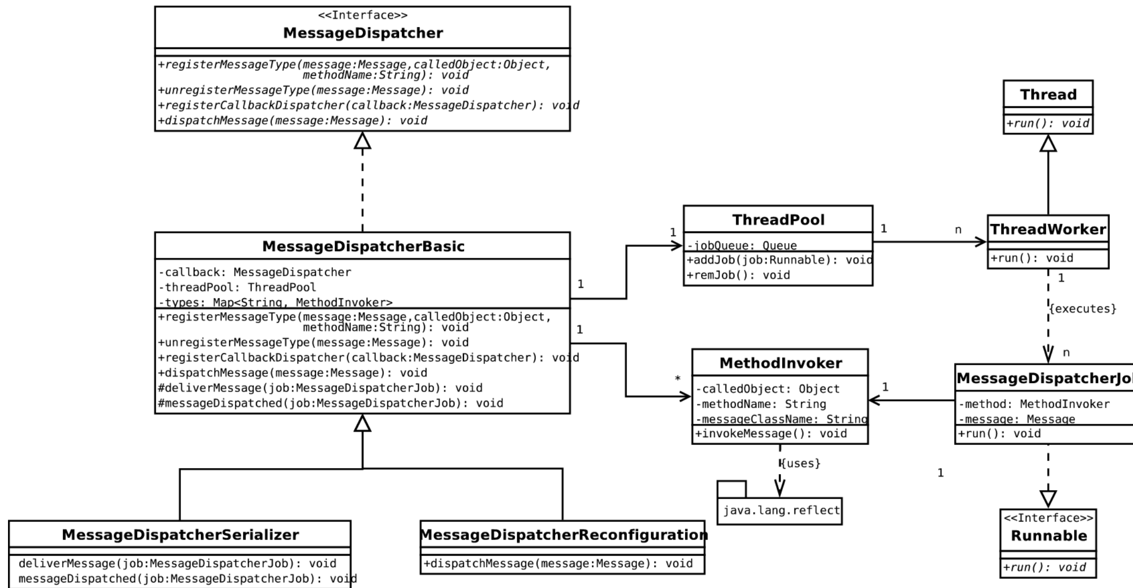


Figure 8.14: Message Dispatcher Class Diagram

The resending mechanism after a timeout implemented will retry to send the message after a certain time and allow the layer to finish the reconfiguration.

The diagram shows how the ReconfigurationManager is able to discern if a message destination is within an interval which is being recovered (actually failed).

MessageDispatcherSerializer : (See Figure 8.17). Messages arrived to this layer are related to transactional semantics and, therefore, concurrency issues. A naïve solution would be to synchronize each message handler in such a way that the code executed within a message handler will be done atomically. This would deal to a high overhead when trying to handle several message concurrently as long as each message introduce a delay within the rest, including when executing the same code for different objects.

Our solution is based on serializing only those messages which are related. We consider that two messages are related if they are sent to the same identifier. This way, we allow concurrent message handler execution when accessing different objects. If two messages are related, the message dispatcher enqueue the next request till the previous one has finished. As we will show in Chapter 9, this mechanism introduces a very little overhead when dealing with high concurrent load, due to a small necessary synchronized part of code .

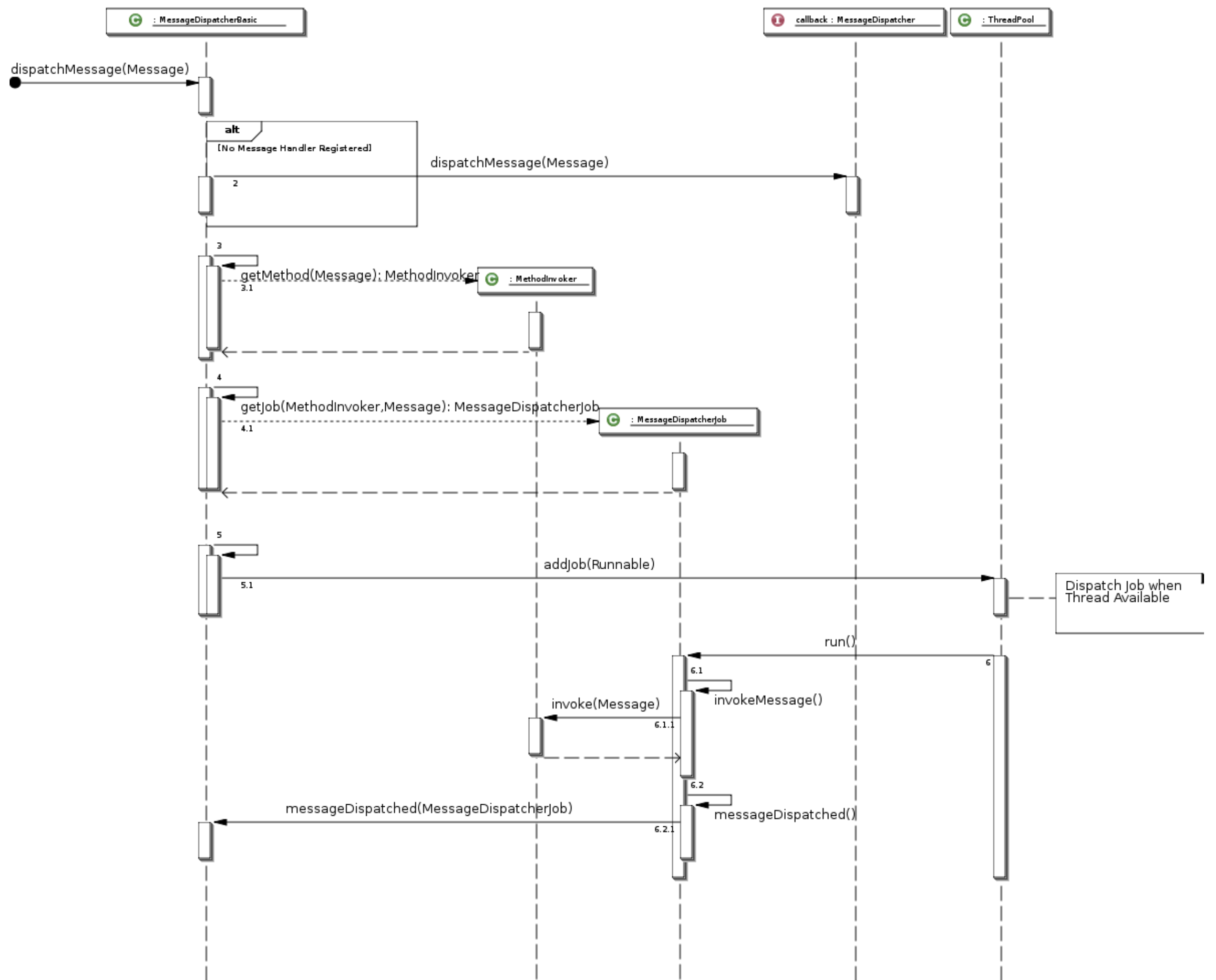


Figure 8.15: Message Dispatcher Sequence Diagram

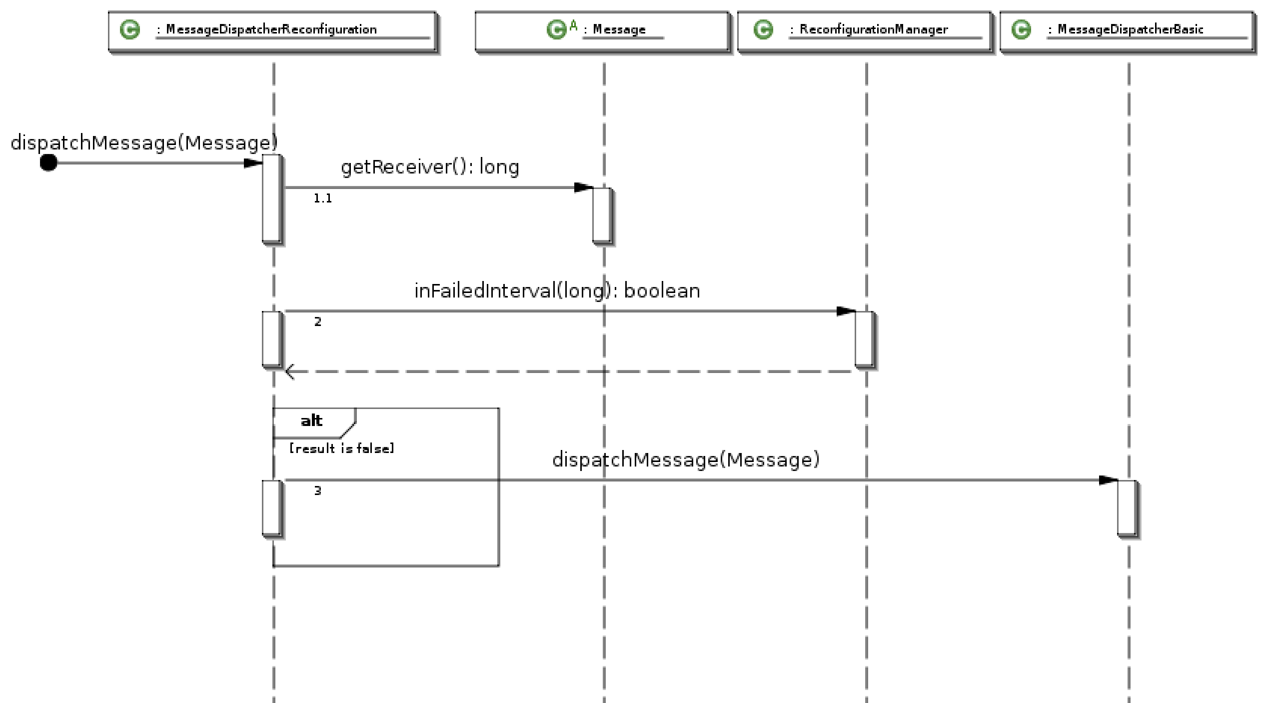


Figure 8.16: Message Dispatcher Reconfiguration Sequence Diagram. This diagram shows the reimplementation of the dispatchMessage method inherited from the MessageDispatcherBasic class. If the destination identifier is within a failed interval, the message is discarded to avoid inconsistent responses.

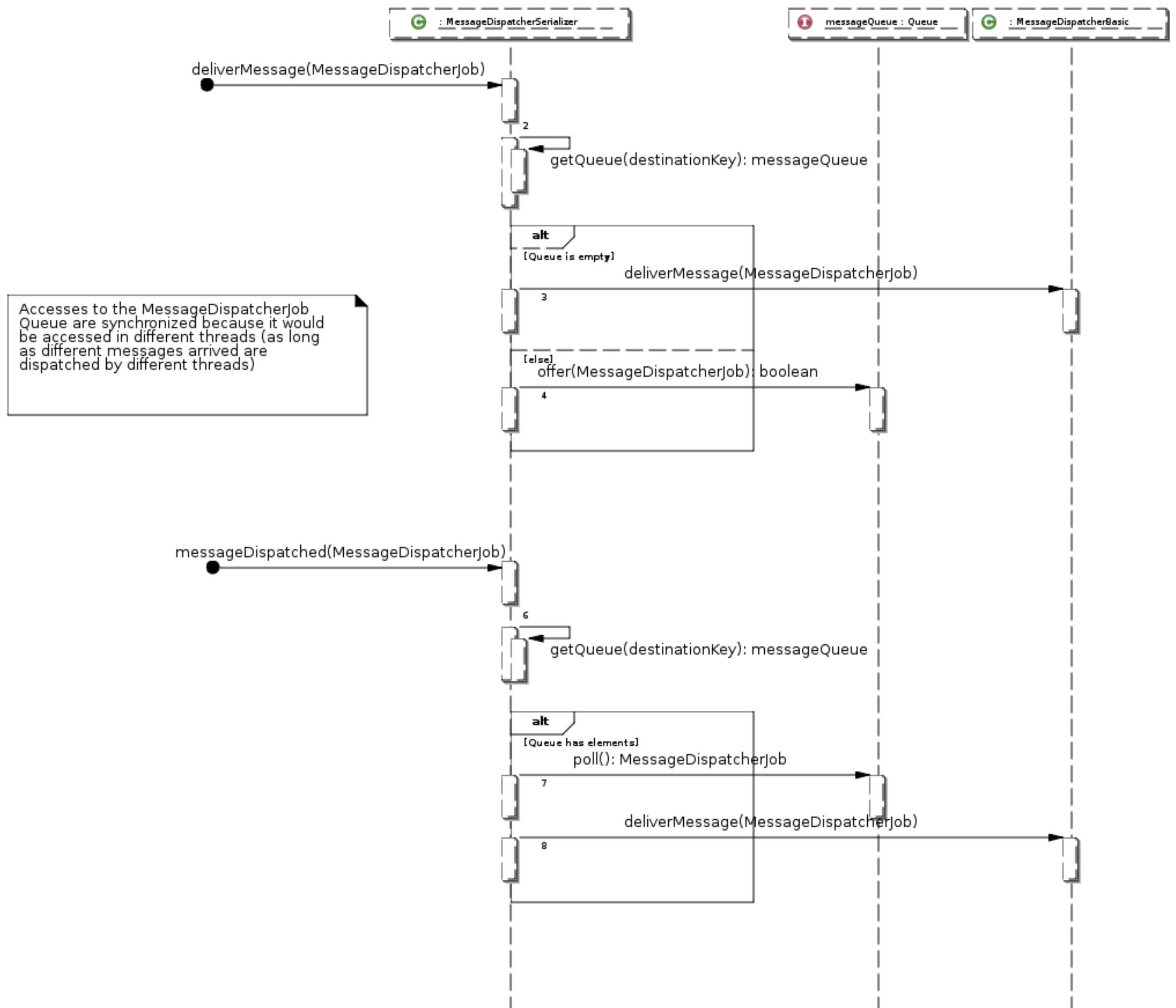


Figure 8.17: Message Dispatcher Serializer Sequence Diagram. Reimplementation of two methods provided by the basic message dispatcher. Both deliverMessage and messageDispatcher are different methods although they are presented within the same sequence diagram.

8.5.2 Synchronous Communications over asynchronous primitives

The main difference between them are the message handling done at the sender node:

Asynchronous communication : once the node delivers the message to the network, it continues with its execution. Further messages will be dispatched by any message handler registered at the message dispatcher.

Synchronous communication : once the node delivers the message to the network, it waits till the other node responses to this concrete messsage. Once the node receives the response it continues with its execution.

We have implemented our synchronous route primitive relying on the asynchronous route primitive. Assume that each message has an identifier constructed by some mechanism and it is unique within the system.

The sender will route asynchronously the message to the receiver. Thereafter, it will wait for a unique object stored within a hashmap indexed by this identifier. The receiver node, after it has received and processed the message, will route asynchronously a new message with the same identifier. Upon the reception of the message, it will check if a thread is waiting for that message querying the previous hashmap. If there was a thread waiting, it passes the message to waiting thread and notify that it is able to continue its execution.

Figure 8.18 and Figure 8.19 shows this mechanism.

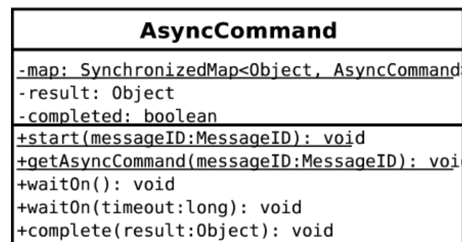


Figure 8.18: Synchronous Communication Class Diagram. This class provides two basic static operations (underlined) to start and get a previously created AsyncCommand Instance. The waitOn() and waitOn(long timeout) are operations to wait till the response is received with and without an spceified timeout). The complete method is used upon the reception of a message to wakeup a thread waiting on this object.

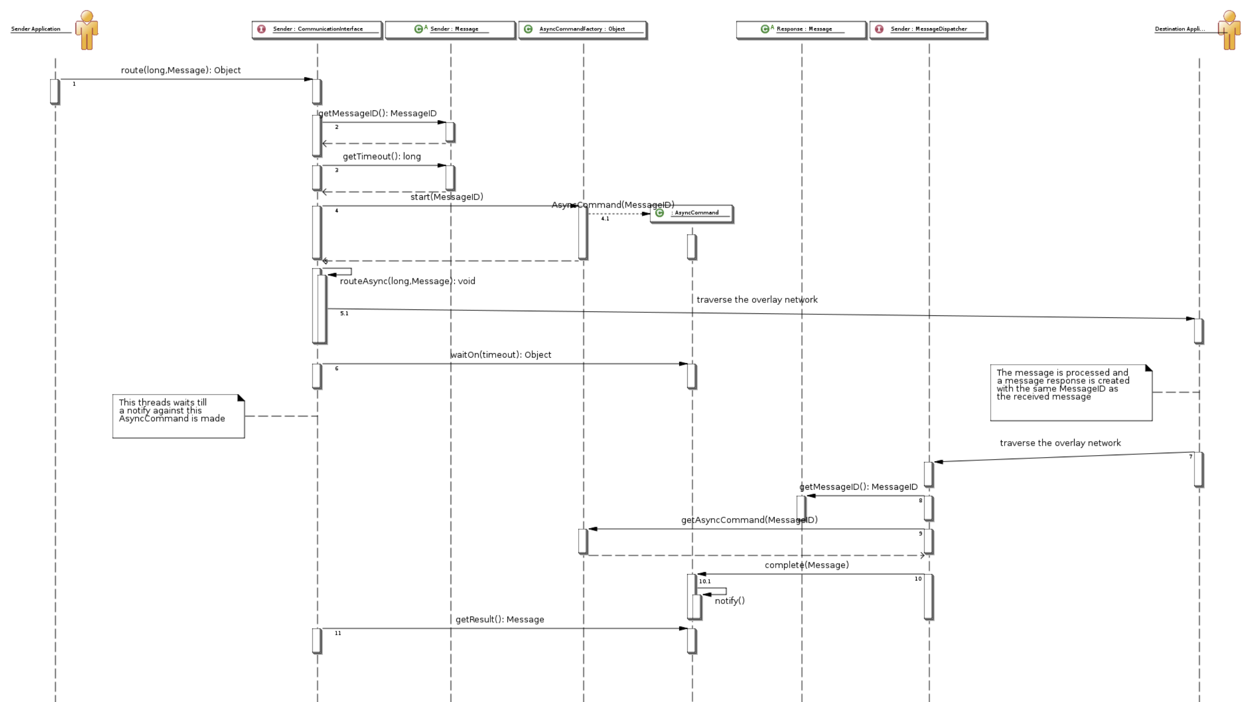


Figure 8.19: Synchronous Communication Sequence Diagram. This diagram shows how the thread responsible to send the message waits till the response arrives. Thereafter, the thread is woken up and is able to continue its execution taking into account the response.

Chapter 9

System Evaluation and Characterization

Throughout this chapter we introduce the experiments executed to evaluate the performance of our system. In Section 9.1 a discription of the execution environment is presented. In Section 9.2 a quantitative perfomance evaluation is shown. Finally, Section 9.3 presents a qualitative comparison of our algorithms against the previous surveyed systems.

9.1 Execution environment

The experiments were executed in a 70 nodes cluster with a Gigabit Ethernet connectivity. Each node is composed of two Intel(R) Xeon(TM) CPU 2.80GHz slots and 2 Gb of shared memory. Each node has a local scratch zone within which accesses have high performance compared with the common scratch zone used to store the results of the experiments.

This cluster is managed by a queue system developed by Sun and called *N1 Grid Engine* which allows users to send jobs to be executed within the less loaded node available. The cluster is called *ferrer* and it is located at the *Laboratori de Càlcul d'Arquitectura de Computadors (LCAC)*.

This way, our experiments were based on executing several jobs at the same time executing our CMS system. Each time a new job was allocated within a node, the first step was to join the overlay through a known node already existing within the overlay ring in order to join and evaluate our assumptions.

The decision of firstly test our assumptions on a cluster instead of on a more real scenarion (such

as PlanteLab [61]) to control more extensively the behaviour of our system without concerning about transmission delays or other unexpected behaviour of planetlab nodes. This way, we could drive controlled experiments and characterize more precisely our system.

9.2 Evaluation

We have divided our experiments in two parts. The first one is the evaluation of our Transactional Layer in terms of the requests throughput and time responses depending on concurrency issues. The second part is a simple theoretical evaluation of our Mutable Consistent Layer in terms of the probability of breaking our assumptions and, therefore, returning incorrect or stale data and its associated cost.

9.2.1 Transactional Mechanism Evaluation

Our transactional mechanism was designed for performing efficient lock and unlock of objects while assuring the mutual exclusion properties. This way, our experiments were driven in this sense: evalute the performance of our decisions compared with the underlying DHT infrastructure without the mutual exclusion mechanism.

The idea is to measure the overhead and lock grant response times taking into account the number of requests per second that a single node could handle and the identifier against which the request is made.

Taking into account our assumption, we expect that when executing different lock requests against different objects (and thus with different identifiers) the response times should be similar to the DKS results as long as they are not conflicting objects. In the other hand, requests against the same object would drive to accumulative response delays due to the queueing of requests.

9.2.1.1 Experiment setup

The first step to perfom such experiments was to set up a 50 nodes ring overlay with pseudorandom node identifier selection. For that purpose, we execute one *bootstrapping node* with a known node identifier (node identifier 300) and within a fixed cluster node (ferrer-80). The rest of nodes which will build the overlay will join it through this well-known bootstrapping node with a pseudorandom node identifier. This way, we assume that nodes are spread evenly arround the ring and none of them would be a bottleneck due to message forwarding.

We perfomed 50 rounds of this setup, increasing sequentially the number of requests per seconds in each round this way: each node performs one request per second against the bootstrapping node for locking and releasing an object in such a way that in the ith round there will be i nodes performing one request per second. This way, the bootstrapping node will handle i requests per second. We mesure the load on the bootstrapping node as long as we assume that requests made to different nodes are treated independently.

These 50 rounds were executed within three different scenarios in the following way:

Contention : each second, nodes perform a request to lock and unlock the object against the same identifier (identifier 100 belonging to the bootstrapping node interval responsability). Thus, we expect to see that the more nodes performing requests, the more long will be the delay to achieve the lock due to our queueing requests mechanism.

Non-Contention : each second, nodes perform a request to lock and unlock the object against different identifiers. Assume each node is labeled in the range $[1 - 50]$, then the nodes perform the requests against the object identified with the identifier 100 + *label*. This way, each node will perform requests against different identifier. Thus, we expect to see that the more nodes performing requests, the time to serve the request would be similar for all of them as long as each object is independent of each other.

DKS : each second, nodes perfom a request to get and put the object against different identifier using DKS native mechanism to manage their DHT. It has no sense to measure its behaviour against the same object as long as DKS does not handle mutual exclusion and thus, the results would be very similar. We will use these results as a basis against which compare our mechanism.

Finally, each round is executed during 15 minutes. We drop off the first 5 minutes of the execution In order to stabilize results.

9.2.1.2 Results

Figure 9.1 shows the median and the standard deviation of the results for each round and scenario. We can see how the Non-Contention and DKS results are similar and highly lower than the Contention scenario. This asserts our assumptions that the more requests per second against the same object, the response times are increased due to the queueing of requests. The most interesting result within this figure is the fact that each request is served within a bounded period of time independently the requests performed asserting the liveness property of our algorithms. It is the consequence of queueing the requests in a FIFO queue.

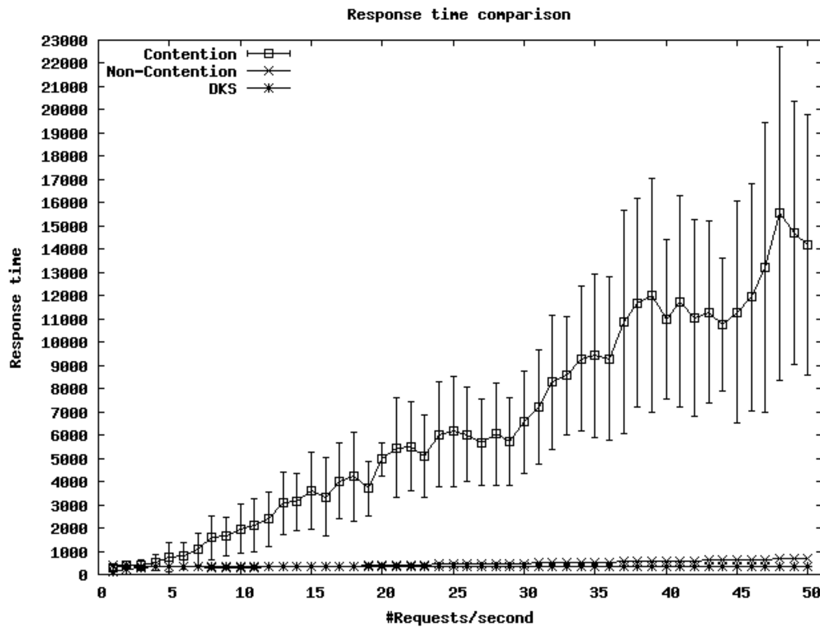


Figure 9.1: Response Time vs Requests rate Comparison

Figure 9.2 shows a zoom of the Non-Contention vs the DKS scenario. We can see that our mechanism imposes a little overhead compared with the DKS as a penalty to achieve the object in mutual exclusion. Despite that, this overhead is only observed when increasing the number of requests to be handled to more than 15. This is due the synchronization variable used by our message dispatcher to deliver the messages in order as explained in Section 8.5.1.

Figures 9.3 and 9.4 are results regarding the throughput comparison of our three scenarios against the requests per second. As expected, our mechanism is able to handle each request per second within the same second in the case of Non-Contention. In the case of Contention, the number of response which is able to handle is stabilized to 4 requests per second (more or less). The figure enlarged shows how the more requests against the same object are made, the less throughput is achieved due to the overhead of updating the state associated to that object.

As the results show, our mutual exclusion mechanism imposes a little overhead when dealing with independent objects compared with the simple DKS approach which does not provide our constraints. In the case of contention, we can see that our solution scales well when dealing with less than four requests per second against the same object. We must take into account that contention translated to our *Currency Management System* means that several transactions are made against the same account (the same user is paying or receiving a payment at the same time). Considering this situation, we do not expect to have such number of concurrent requests against the same account.

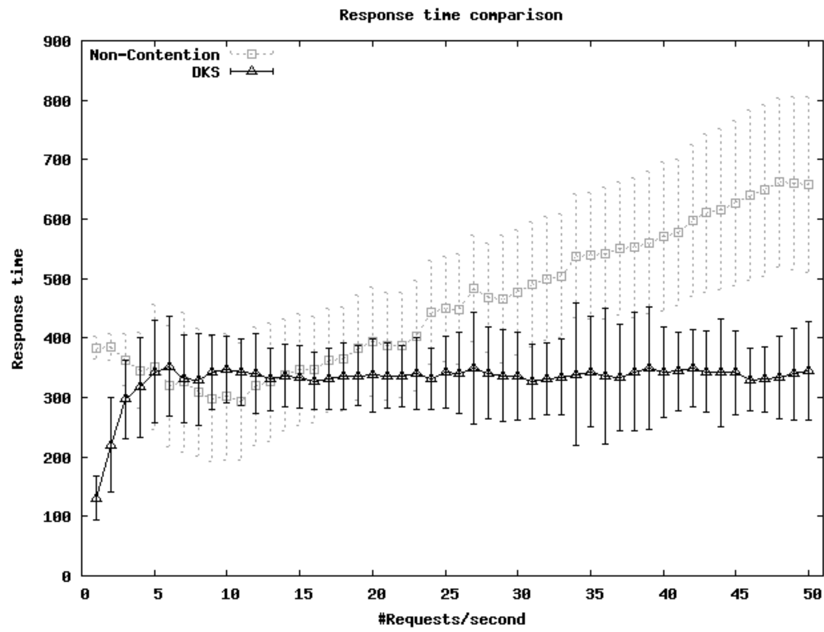


Figure 9.2: Reponse Time vs Requests rate zoom comparison

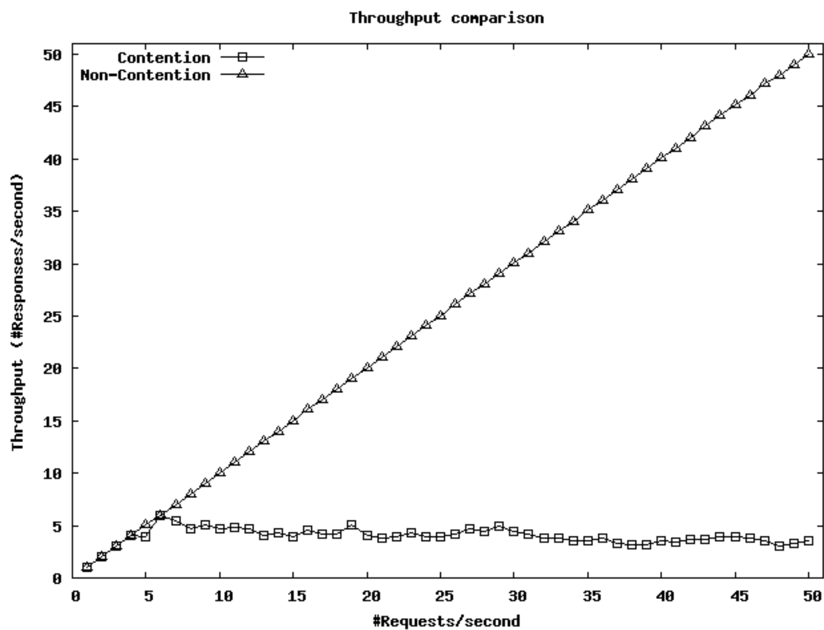


Figure 9.3: Throughtput vs Requests Rate of our system with and without contention. That is, performing requests against different objects and against the same object respectively.

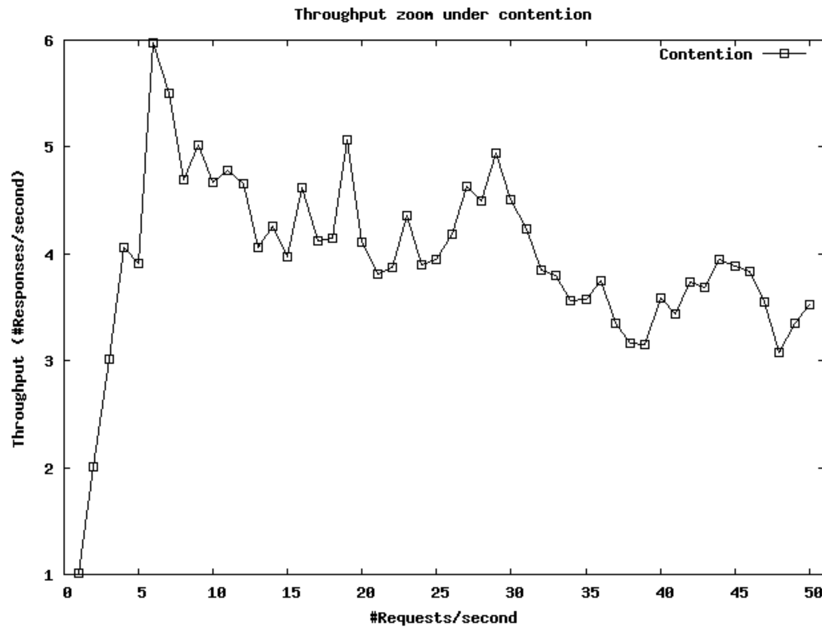


Figure 9.4: Throughput vs Requests Rate of our system with contention. The zoom is shown in order to show more concretely how the more requests are made against the same object, the throughput decreases.

9.2.2 Consistency Mechanism Evaluation

Our transactional mechanism was implemented on top of our enhanced DHT abstraction relying on its high probability of returning the most up-to-date data stored under a given key. While returning up-to-date data in case of joins and leaves is assured by the *lookup consistency* property provided by DKS and routing each requests to the single responsible node for a given key, in the case of failures this is not true. Our aim was to provide low probability of recovering stale data in case of failures.

To measure our enhancements against the DKS approach, we consider a ring overlay where nodes may fail arbitrarily. Consider that the probability of losing a replica update is p. If a replica is not updated, we consider that it stores an *stale* item. For simplicity, we do not consider the reason for that replica update failure (it may be due to message lost or replica node failure).

DKS recovery : when a node detects that its predecessor has failed it must recover that failed interval. For that purpose it only asks the first replicated interval to recover the items. If the interval is recovered, those items might be stale data with a probability of p as long as it does not asks each replicated interval (maybe the most up-to-date data is within the last replicated interval). It does not take into account the number of replicas within the system.

Consistent recovery : when a node detects that its predecessor has failed it must recover that

failed interval. For that purpose it asks each replicated interval to recover the items and select the must up-to-date, or in other words, the items with a higher associated timestamp. Thus, the probability of recovering stale data is the probability of that each replicated interval loses the last update. Therefore, the probability is p^k where k is the number of replicas within the system.

Figure 9.5 shows how the probability of recovering stale data with DKS is constant independently of the number of replicas within the system. In our solution, the more replicas within the system the less probability of recovering stale data is achieved. Moreover, our system is able to deliver a high probability of recovering up-to-date data when dealing with a high failure probability (0.5) setting up the system with 8 replicas.

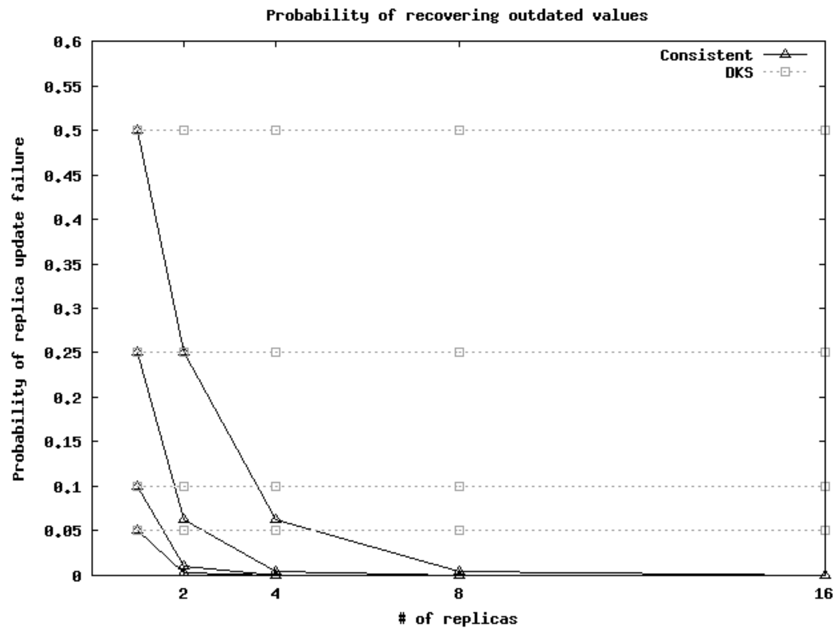


Figure 9.5: Probability of recovering stale data depending on the number of replicas within the system.

Nevertheless, our enhancements come at a cost of sending several messages to k replicas and merge different items from different replicated interval. As long as DKS only asks one interval, the first interval recovered is considered as correct and stops the recovering process. In our case, we must ask each replicated interval and waits till every response is received in order to select the item with the highest timestamp.

Figure 9.6 shows that, while the DKS recovery times remains constant independetly of the number of replicas used, the more replicas used in our solution the more time it takes to make a decision on which item is the newest one.

This experiment was based on the previous presented setups. We set up a 64 nodes ring overlay with a random identifier assignment to evenly distribute nodes along the ring. Each minute, we stop one node and measure the time taken by its successor to recover the failed interval. This procedure was done for every node within the ring. This round was repeated for different replica setups and the median for each round is presented in Figure 9.6

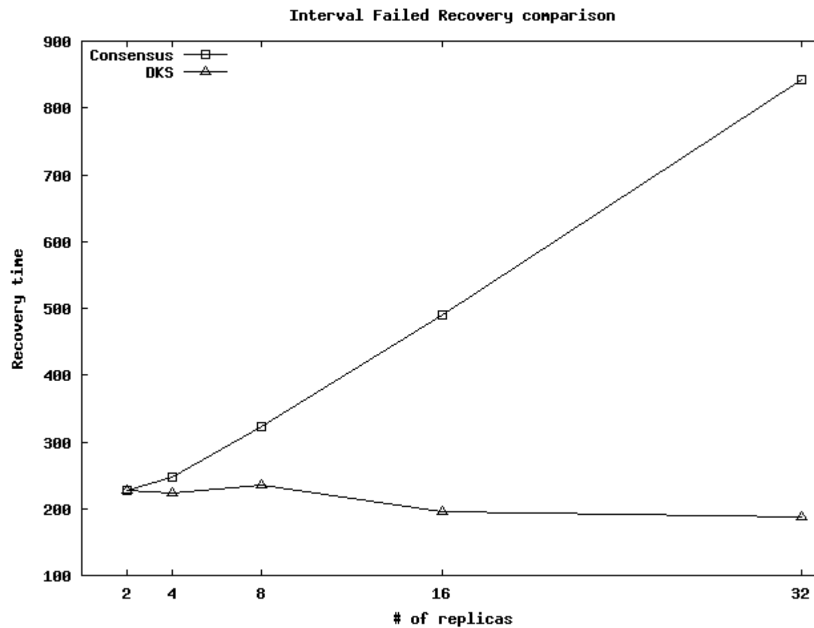


Figure 9.6: Interval recovery comparison taking into account the time frame from the detection of a failed interval to the complete recovery of the interval.

9.3 System comparison

Throughout this section we provide a simple comparison of our system against two of the systems presented in Part I regarding consistency as well as mutual exclusion mechanisms.

Regarding consistency mechanism, our solution is more efficient in terms of message complexity that the one presented by Etna [48] when dealing with joins and leaves as long as we only need to send messages against one node (the joining or leaving one) whilst Etna needs to reconfigure its state by contacting each node within the new replica set. In the case of failures our performance is similar as long as our solution drives the system to contact each of the replicas as in the case of Etna. This improvement is, to a great extent, thanks to the symmetric replication approach and the lookup consistency mechanism provided by the underlying KBR Layer.

Regarding the mutual exclusion issue, our solution is more efficient in terms of messages sent that

the ones presented in Section 4.4 as long as we focus our solution in a central coordinator approach which only needs two messages (request/response) to grant the request in mutual exclusion. By the other way, other mechanisms were based on a distributed permission based approach which drives to a number of messages proportional to the number of replicas within the system. Moreover, we provide certain guarantees such as FIFO delivery which distributed permission based are not able to provide. Nevertheless, our system is prone to long delays to serve requests in case the central coordinator for a certain interval fails (as long as the new responsible has to recover the interval failed).

Chapter 10

Project Plan and Economic Evaluation

The duration of the whole project, supposing a full-time work of 7 hours a day, 5 days a week, is shown (with the Gantt diagram) in Figure 10.2.

All the work is clearly divided in three different phases, namely survey of related work, development of the system itself and the proposal documentation. Notice that the development process began with our enhancements of the DKS DHT as long as it is the basis for the rest of upper layers.

Besides, the critical path is further enlarged due to the delay introduced by the developers of the underlying p2p infrastructure. As long as our system is developed on top of the DKS DHT, we were waiting for a newer release they have planned to publish in the early December in order to take profit of newer advantages. This advantages ranged from more polish and documented code to more extensibility capabilities of their software.

This release is not still published so our decision was to base our code in the older version (which stands as a proof of concept of DKS algorithms). This way, the lack of documentation of their old DKS and this waiting for the new release introduced a huge delay before the development process began.

Even that, there is some work that can be parallelized. Although they are all separated in multiple roles, all the project has been done by a single person, which makes it difficult to know if the different role timings have been strictly followed, and adds some indeterminism in the realization of it, due to the constant switching between the many open tasks.

Figure 10.1 and Table 10.1 indicates the duration and costs of the different tasks of the project as well as the role assignment respectively. To calculate that, an estimation of the prices in the computer sector has been used (from a study made during 2006 by AETIC [62]).

WBS	Name	Start	Finish	Work	Duration	Slack	Cost	Assigned to
1	Background and Related Work	Oct 2	Nov 3	25d	25d	51d	11,600	PM
2	Requirements and Specification	Nov 6	Nov 17	10d	10d	158d 5h	2,960	AF
3	**System Construction**	**Dec 15**	**Mar 5**	**101d**	**57d**	**52d 5h**	**30,256**	
3.1	**Consistency Module**	**Dec 15**	**Feb 5**	**37d**	**37d**	**72d 5h**	**10,832**	
3.1.1	Specification	Dec 15	Dec 25	7d	7d		2,072	AF
3.1.2	Design	Dec 26	Jan 15	15d	15d		5,400	AT
3.1.3	Implementation and Tests	Jan 16	Feb 5	15d	15d	52d 5h	3,360	AP
3.2	**Transactional Module**	**Dec 26**	**Feb 26**	**37d**	**45d**	**57d 5h**	**10,832**	
3.2.1	Specification	Dec 26	Jan 3	7d	7d	8d	2,072	AF
3.2.2	Design	Jan 16	Feb 5	15d	15d		5,400	AT
3.2.3	Implementation and Tests	Feb 6	Feb 26	15d	15d	52d 5h	3,360	AP
3.3	**Banking Module**	**Jan 4**	**Mar 5**	**27d**	**43d**	**52d 5h**	**8,592**	
3.3.1	Specification	Jan 4	Jan 12	7d	7d	16d	2,072	AF
3.3.2	Design	Feb 6	Feb 26	15d	15d		5,400	AT
3.3.3	Implementation and Tests	Feb 27	Mar 5	5d	5d	52d 5h	1,120	AP
4	System Evaluation	Mar 6	Apr 16	30d	30d	52d 5h	8,760	AP, AT
5	**Documentation**	**Nov 6**	**Jun 19**	**116d**	**161d 5h**		**42,818.67**	
5.1	Related work	Nov 6	Dec 15	30d	30d	51d	13,920	PM
5.2	Consistency Module	Feb 27	Mar 22	18d	18d		5,904	AF, AT
5.3	Transactional Module	Mar 23	Apr 11	16d	13d 2h		4,693.33	AF, AP, AT
5.4	Banking Module	Apr 11	Apr 30	16d	13d 2h		4,693.33	AF, AP, AT
5.5	Evaluation	Apr 30	May 24	18d	18d		5,256	AP, AT
5.6	Conclusion and Review document	May 24	Jun 19	18d	18d		8,352	PM
6	Presentation	Jun 19	Jun 28	7d	7d		3,248	PM

Figure 10.1: Task duration and cost. The cost of each task depends on its duration as well as the cost per hour of its associated roles.

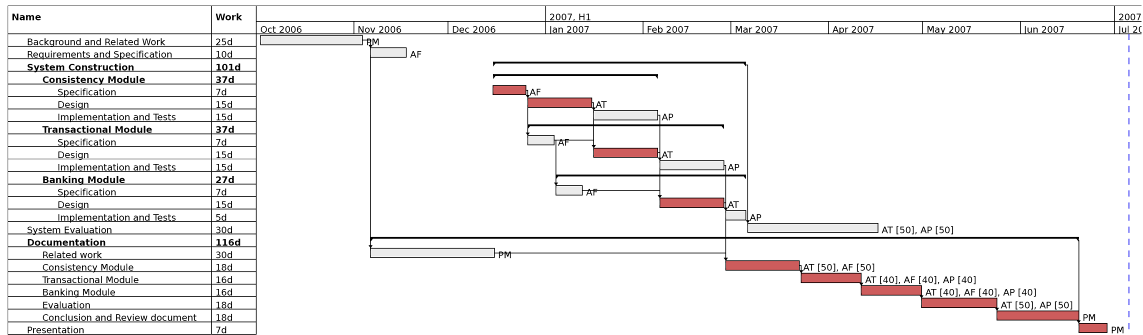


Figure 10.2: Project Planification Gantt Diagram. The critical path is highlated.

Role	Alias	€ h
Project Manager	PM	58
Technical Analyst	AT	45
Functional Analyst	AF	37
Programmer Analyst	AP	28

Table 10.1: Costs by roles

The only cost that has not been taken into account is that of the software and the hardware. The first, as we have entirely used free software, it has no cost, while for the later a Laptop with an Intel Centrino of 1.8Ghz, 1GB of RAM and less than 50 GB of HD has been used. This machine has a current cost in the market of approximately 900 €. Taking into account a 4 years lifetime for the laptop and the duration of the current project (9 months approx.), it implies an amortization cost for the laptop of approximately 169 €. The hardware used to perform the experiments were provided by the Computer Architecture Department Lab for free so no extra cost is added for using their cluster.

Thus, taking into account the different human resources working in different roles as well as hardware and software resource costs, the overall cost of the whole project rounds the 99811 €.

Chapter 11

Conclusions and Future Work

In the last few years, the resource allocation problem within internet-scale Grid deployments has been addressed using economic models. Whilst lot of research has been done to demonstrate the validity of these models, the lack of mechanisms to control the resource usage amongst several users has slow down real deployments. One of the main objectives within Grid4All is the deployment of real market based resource allocation by enabling users to share their resources in exchange for real money.

Within this context, we have developed a distributed banking service for the Grid which enables users to perform and receive payments for resources usage and sharing without incurring in the cost of real payment mechanisms, i.e avoiding taxes.

We have implemented our prototype by developing an enhanced DHT which provides stronger guarantees (compared with current DHTs implementations) such as:

Consistent Storage : by modifying DKS in order to assure that every *get* operation returns the last *put* operation through the responsible node for a given object regardless the inherent dynamism of p2p overlay networks.

Better Concurrency Guarantees : by allowing clients to acquire an object in mutual exclusion avoiding possible conflicting updates. We take profit of the underlying enhanced DHT infrastructure in order to simplify the algorithms and to ease the state's transfer associated with a locked object.

Transactional semantics : by enabling applications to perform a sort of database transactions when dealing with operations which need to be performed with ACID properties.

The layered architecture enables us to exchange whatever component in the near future and use it

independently. Moreover, the Mutual Exclusion technique on top of a DHT might be used stand-alone as a sort of *lock service* in such a way that whatever system defined component needed to be accessed in mutual exclusion may be accessed through this service by locking its associated object. For instance, assume a cluster of computers identified by its IP address and applications which need to execute applications on some nodes in mutual exclusion (no more than one application per node). The application might ask for the lock of the object associated with the IP address of the node and release it once the application has ended. The rest of applications willing to access this node will wait till the application has ended.

The results presented encourage us to continue with the development of our prototype by enhancing the message serializer dispatcher component which is the main cause of the overhead generated.

Another open issue is the case when a client executing a transaction fails after committing some of the objects but not all of them. This case breaks the atomicity property as long as some updates are performed and some others not. Nevertheless, this would be solved if we use the underlying DHT to store the transaction objects as well. This way, if a node fails when committing a transaction, the new responsible node for the given transaction would be able to end up this transaction regardless the committed objects. However, we have to investigate if the cost associated to inserting a transaction within the DHT worths, taking into account the low probability of client failure within a limited bounded time during which the transaction is committed.

Whatever enhancements might be introduced to our algorithms, we expect to be working on the following checkpoint list in the near future:

Fractal component integration : The componentization of our Currency Management System in *Fractal components* [58] is necessary in order to integrate it with the rest of Grid4All components as long as the deployment of applications within the Market Framework will be done by means of this composition technique. Moreover, we must still define how the Market Framework service will be accessed in order to develope the *CMS Gateway Interface Layer*.

Adaptation to Niche (former DKS) : Adapt and integrate our solution to the underlying overlay infrastructure of Grid4All. Despite our solution is based on DKS, a new version of this p2p middleware will be released in the new future with the name of *Niche*.

GMM service integration : Discuss about integrating our Currency Management System inside the *Grid Market Middleware* (GMM) [63] we are developing within the *Computer Networks and Distributed Systems* research group. The GMM is an economic based grid resouce management middleware which offers an open platform that supports the implementation of

diverse economic based resource allocation mechanims, offering a set of basic mechanisms that facilitate its implementation.

More long term development issues will be the integration of our payment mechanism with a real one in order to carry out the real payments once the user wants to withdraw its current virtual currency by another real currency (for instance, PayPal).

Part III

Appendixes

Appendix A

Conventions for the notation of algorithms

Throughout the document, we used a custom algorithm notation in order to specify the messages sent by different nodes. Within this appendix, we specify the notation convention in order to clarify the semantics used.

Each component has its own specific methods with its own specific parameters. Assuming that the signature of the different operations for each component is known, we use the following notation to denote a call to an API method depending on wether it returns some value or not:

component.methodName(parameters)

result ← ***component***.method(parameters)

Notice that these calls are made local to the node. In other words, the computation performed with these calls are local and no messages are sent through the network to execute them.

Regarding the KBR Layer, a special notation has been used to ease the understanding of the message passing interface used. This way, we use a similar notation to that of the RPC calls, specifying the destination node id, the remote procedure name and its parameters. The RPC notation is similar to our system in the sense that we have a handler for each message. In this sense, we consider the message type as the procedure name and the message handler as the logic executed by the receiver of the RPC call.

In Algorithm A.1, we show an example containing the three primitives provided by the KBR layer. We have adapted them to a remote procedure call in the following way:

restrictedBroadcastAsync : executes the *remoteProcedureName* procedure on each node belonging to the interval specified by parameter and which begin with node *i*.

routeAsync : executes the *remoteProcedureName* procedure on the node which is responsible for identifier *i*.

route : executes the *remoteProcedureNameSync* procedure on the node which is responsible for identifier *i* and stores the result of the remote call in *response*.

Notice that each remote procedure specification has a label *from* to identify the source of the RPC in case the procedure needs this information. There are special cases where a RPC needs to return different values to more than one node. In this case, the return statement is represented as follows:

return *result* **toNode** *nodeIdentifier*

In case the remote procedure only return only one value, it is sent by default to the invoker of the remote procedure.

Algorithm A.1: Algorithm Example

```
1:  procedure component.localProcedureName1(parameters)
2:      response ← component.localProcedureName2(parameters)
3:  end procedure
4:  procedure component.localProcedureName2(parameters)
5:      restrictedBroadcastAsync k.remoteProcedureName(interval, parameters)
6:      routeAsync k.remoteProcedureName(parameters)
7:      response ← route k.remoteProcedureNameSync(parameters)
8:      return response
9:  end procedure
10: procedure k.remoteProcedureName(parameters) from i
11:     local computations
12: end procedure
13: procedure k.remoteProcedureNameSync(parameters) from i
14:     local computations
15:     return result
16: end procedure
```

Appendix B

User Guide

Throughout this appendix, we introduce how to execute and test our system in different nodes (or under the same machine) as well as introduce different parameters which can be modified within the configuration file in order to modify the behaviour of our system.

B.1 Basic Configuration

The configuration implemented within our system is based on a simple properties file. These properties are named by a property name and have an associated value. For example:

```
test.parameter param
```

This way, to access this property when implementing some module we may access it through the *Environment* class in the following way:

```
String value = Environment.getProperty(‘‘test.parameter’’);
```

Now we have a string variable representing the value *param*.

We have implemented the configuration component in such a way that when initializing the system we read a default configiguration file named *config.properties* which must be within the same directory where the system is executed. If no file is present, the initialization process will fail.

Nevertheless, each property defined within a configuration file may be overriden by command line argument (including the configuration file used which is indexed by the property *config.file*) in the following way:

```
> java -cp cms.jar ClassName property.n1 value1 property.n2 value2 ...
```

Table B.1 shows all the configuration properties with its allowed values and a brief description. This would help the user to modify the configuration file and adapt to its necessities. There are other properties not introduced here which were created in order to test and tune our system and may rest unmodified to avoid unexpected behaviour.

Property	Allowed Values	Description
dht.replicationDegree	$2^k \ \forall k \in N$	Informs the system of the number of replicas which will be within the system
dht.failedIntervalType	dks/consensus	Informs the system whether to use the dks default recovery mechanism or our technique respectively
bind.ip	string	Hostname or ip (x.x.x.x) to bind on the local machine
bind.port	$p \in [1024, 49151]$[1]	Port number to bind on the local machine. If port is equal to 0, a random free port is used
bind.idPolicy	fixed/random	Informs the system whether the user will specify the id within the ring or it may be assigned randomly
bind.id	$i \in [0, 4294967295]$[2]	ID of the node within the ring (only applicable when bind.idPolicy is fixed)
boot.ip	string	Hostname or ip (x.x.x.x) to use as bootstrapping node to join the ring
boot.port	$p \in [1024, 49151]$	Port number where the bootstrapping node listens to
boot.id	$i \in [0, 4294967295]$	ID of the bootstrapping node within the ring
bootNode	boolean	True if it acts as a bootstrapping node. False otherwise.

Table B.1: Configuration properties provided to change the basic behaviour of our system.
[1] *This range is specified by the IANA. Special rights are not necessary to bind to these ports*
[2] *This range is the current identifier space of the underlying overlay network*

B.2 Execution

The only one requirement to execute the system is, basically, the Java Virtual Machine 1.5.0 from Sun. Other virtual machines will fail due to the use of specific features such as typified structures or enumerations provided by the Sun JDK. We have packaged our system in such a way that each library requirement is packaged within a single jar file (thanks to the onejar application [64]). As long as our system is built upon the Java virtual machine, it is operating system independant so it might be executed in whatever environment with a java virtual machine implementation (i.e. Linux, Windows, etc).

For simplicity when building the overlay network, we have divided nodes in two different roles:

Bootstraping node : this node will create the overlay network and it will be the first to join it. The rest of nodes will join the overlay through this node.

Joining node : this node will join an existing overlay node through a well known bootstraping node.

This way, in the *package* directory of the CD included with this document, there is a single jar file called **cms.jar** and two sample files called *boot.properties* and *join.properties* which initialized the system depending on the role played by the node (bootstraping or joining node).

Despite it was not a primary component for this project, we have implemented a simple banking command line interface by means of which the user could interact. This command line interface has been implemented in order to carry out our tests and to exemplify how the Currency Management System works.

B.2.1 Localhost Setup

This section shows how several nodes may build an overlay within the same machine (that is, binding to the localhost interface).

B.2.1.1 Bootstrap node

The first node to boot up is the bootstraping node. For this purpose, the user must execute the following code:

```
java -jar cms.jar config.file boot.properties
```

This will raise the following text within the console where the above command was executed:

```
        **********************************************
        * Welcome to the Currency Management System *
        **********************************************

        Credentials for sysadmin: sysadmin
[BOOT] NodeID: 0
[BOOT] WebID:  http://127.0.0.1:3435/info/0/0/0/2169371122624172466
        Usage:
                openAccount [Credentials]
                closeAccount [ID] [Credentials]
                queryAccount [ID] [Credentials]
                depositFunds [ID] [amount] [sysAdminCredentials]
                transferFunds [srcID] [dstID] [amount] [Credentials]
                exit
```

The lines beginning with the *[BOOT]* text are information referring the current node being executed. They specify the NodeID assigned as well as a web interface provided by DKS which provides information about current neighbours and its routing table. Moreover, increasing by one the port used in the web interface, a new web page interface is provided by us showing current storage information within this node (number and size of items which the node is responsible to store and replicate as well as information about the stored objects themselves). We encourage the reader to use those interfaces to navigate the overlay network once some accounts have been created and modified.

Our web interface shows the following information once four accounts have been modified and a transfer of Funds have been performed (for each account, its associated transactional object and state object are stored, See Figure B.1).

B.2.1.2 Joining Node

Subsequent nodes willing to join the overlay will do it through the bootstraping node boot before. It is a simplicity decision, nevertheless each node might join through every node already belonging the overlay network. For this purpose, the user must execute the following code:

```
java -jar cms.jar config.file join.properties
```

B.2. Execution

Besides the same console messages as the boot node, a new message will be shown informing that some items has been transferred to the new joining node resembling this one:

```
[WARNING][org.kth.dks.dks_marshal.ObjectAdapter]
edu.upc.cnds.cms.transactionalLayer.TransactionalLayerImpl.
intervalJoined(TransactionalLayerImpl.java:202): TODO! intervalJoined...
[start:0 end:2422112333]
```

Now the overlay is builded upon two nodes. Subsequent nodes will increase the number of nodes within the overlay balancing the load of managing such stored items.

```
File  Edit  View  History  Bookmarks  Tools  Help
http://127.0.0.1:3436/info/0/0/0/2169371122624172466
DKS DHT API   DIET Agets API   Java 2 Platform SE ...   DAC - LCAC   DAC - Intranet   Engineering Village ...

General Informations (in KB)
----------------------------
Num Items:               8
Num ReplicationItems:    8
Size Items:              9
Size ReplicationItems:   9

Data Stored in the localDHT ([id] timestamp objectClass)
---------------------------------------------------------
Replica Index: 0
Responsability: [start:0 end:0]
-------------------------------------------------
[3414795,4298382091] 2 StateObject State: FREE
[3414795,3414795] 1 TransactionalObject ObjectType: Account StateID: [3414795,4298382091] Object: Account:
        Owner: omer
        Balance: { Current Balance: 7, Reserved Balance: 0 }
        History of transactions:
                DepositTransaction: { srcAccount=50652283, dstAccount=3414795, amount=7, timestamp[1][0] }

[50652283,50652283] 2 TransactionalObject ObjectType: Account StateID: [50652283,4345619579] Object: Account:
        Owner: leandro
        Balance: { Current Balance: 30, Reserved Balance: 0 }
        History of transactions:
                DepositTransaction: { srcAccount=114140560, dstAccount=50652283, amount=37, timestamp[1][0] }
                WithdrawTransaction: { srcAccount=50652283, dstAccount=3414795, amount=7, timestamp[1][0] }

[50652283,4345619579] 4 StateObject State: FREE
[97315196,97315196] 0 TransactionalObject ObjectType: Account StateID: [97315196,4392282492] Object: Account:
        Owner: felix
        Balance: { Current Balance: 0, Reserved Balance: 0 }
        History of transactions:

[97315196,4392282492] 0 StateObject State: FREE
[114140560,114140560] 2 TransactionalObject ObjectType: Account StateID: [114140560,4409107856] Object: Account:
        Owner: xleon
        Balance: { Current Balance: 63, Reserved Balance: 0 }
        History of transactions:
                DepositTransaction: { srcAccount=114140560, dstAccount=114140560, amount=100, timestamp[0][0] }
                WithdrawTransaction: { srcAccount=114140560, dstAccount=50652283, amount=37, timestamp[1][0] }

Done
```

Figure B.1: Simple Web Interface of the Mutable Consistent DHT Layer. The General Information part shows the number and size in KB of items stored as primary replica and as backup replica. The Data Storage part shows each object stored presenting the following information in the same order: ID, Timestamp, ClassName of the object stored, String representation of the object -call to the toString() method-.

B.2.2 Distributed Setup

This section shows how different overlay nodes could build an overlay distributed upon several machines connected by whatever mechanism (internet, LAN, etc.). For that purpose, assume we have *n* numbered in the range [1,n] with ip addresses of the form 192.168.2.*n*. It is an arbitrary configuration usefull to understand the procedure and may be replaced by whatever real configuration. Number 1 will be used by the bootstraping node. The rest of numbers will represent joining nodes.

For the distributed setup, the procedure is the same as in the localhost setup by overriding some configuration parameters to point to the correct interface as long as the configuration files provided binds to the localhost interface.

This way, the command to execute in the case of the bootstraping node is:

```
java -jar cms.jar config.file boot.properties bind.ip 192.168.2.1
```

In the other hand, the command to execute in the case of the joining nodes is:

```
java -jar cms.jar config.file join.properties bind.ip 192.168.2.n
                  boot.ip 192.168.2.1
```

where *n* is the number assigned to each joining node to construct its own IP. The rest of properties defined in the configuration file helps us to build it in an easy way. Nevertheless, the user might want to override some parameters and setup its own configuration. In this sense, there are some parameters that are mandatory in order to be able to build the ring (no matter if they are provided by the configuration file or by command line). These are shown in Table B.2.

Role	Mandatory Properties
Bootstraping Node	bootNode[*true*], bind.ip, bind.idPolicy, bind.id, bind.port
Joining Node	bootNode[*false*], bind.ip, bind.idPolicy, bind.id, bind.port, boot.ip, boot.id, boot.port

Table B.2: Mandatory properties for building the overlay.

B.2.3 Playing with it

Once the overlay is set up in whatever form, the commands shown can be executed in order to perform transactions. Some consideration to take into account:

- the Credentials parameter must be a simple string without spaces. The ID of the account associated will be constructed with the hash of this string.
- the Credentials of each operation is referred to the string used to create the source account of the operation.
- the operation depositFunds may only be executed by the sysAdmin actor of the CMS. In this sense, we provide a simple system administrator credentials which is represented by the string *sysadmin*.

Now, the user can create, modify and delete whatever account from whatever node belonging to the overlay. In other words, users may perform modifications to one account from one node and, subsequent queries from whatever other node, will lead to consistent results. Notice that each account is created with no credits so the first operation to be made is *depositFunds* to one account to create the initial virtual currency.

Appendix C

Glossary

ACID : Acronym for Atomicity, Consistency, Isolation and Durability. Four desirable properties when performing transactional operations in database related systems.

Actor : using UML notation, it is is something or someone who supplies a stimulus to the system. An actor cannot be controlled by the system and is defined as being outside the system. It is often thought of as a role, rather than an actual person. A single person in the real world can be represented by several actors if they have several different roles and goals in regards to a system.

AETIC : Acronym for Asociación de Empresas de Electrónica, Tecnologías de la Información y Telecomunicaciones de España. Its main objective is to promote the TI sector, specially with the added value services.

API : Acronym for Aplication Program Interface. External source code interface provided by every component to deliver its functionalities to the rest of components. More concretely, it is the set of methods, functions or procedures provided by a component.

CMS : Acronym for Currency Management System.

Computer Cluster : It is a group of tightly coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks.

DHT : Acronym for Distributed Hash Table. As its name suggests, it is a hash table which is distributed upon some cooperating nodes.

DKS : Acronym for Distributed *K*-ary System. Peer to peer middleware based on a structured p2p overlay network which offers some interesting properties compared with other DHT based implementations.

GRIMP : Acronym for **GRI**d4All **M**arket **P**lace. It is an structure where the generic components that are fundamental to implement market resource allocations systems are identified. This component is specific for the Grid4All architecture.

Infrastructure : It is, generally, a set of interconnected structural elements that provide the framework supporting an entire structure. In other words, the infrastructure should be understood as external components not provided by the system itself which their use helps the system to to achieve its objectives.

Item : It is the minimum stored object within DHTs. Basically, it is a {*key,value*} pair which helps the DHT to store it at the responsible and replica nodes.

JDK : Acronym for Java Development Kit. Set of programs and libraries for Java based program development.

KBR : Acronym for Key Based Routing. Substrate of many p2p structured overlay networks which is based on delivering messages to a node responsible for a given identifier.

Node *i* : Abbreviation of the sentence *Node wich is responsible for identifier i*. As long as the KBR routes a message to the node responsible for a given identifier and identifier responsability depends on the dynamicity of the system, the system based on the KBR does not matter about which physical node is responsible for an identifier at a given time.

Overlay Network : computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. i.e a p2p network.

P2P : Acroynm for Peer-to-Peer. Overlay network build upon several nodes (peers) which allow to share wahtevet kind of resource to the community.

PKI : Acronym for Public Key Infrastructure. In cryptography, PKI is an arrangement that binds public keys with respective user identities by means of a certificate authority (CA). The user identity must be unique for each CA. This is carried out by software at a CA, possibly under human supervision, together with other coordinated software at distributed locations. For each user, the user identity, the public key, their binding, validity conditions and other attributes are made unforgeable in public key certificates issued by the CA.

Replica : It is a copy of the original object. Replicas are often used for replacing the original one in case the original is not available. It is a technique in distributed system to enable high data availability.

Routing Protocol : Routing protocols allow different computer networks to communicate. Routing protocols specify the set of rules that help the overlay node to pass information among themselves on the topology of the overlay network.

Routing Table : A routing table is a table on a router that is used to store that router's information on the topology of the network immediately around it. It is used to direct forwarding of packets by matching destination addresses in a packet to the network paths in routing table used to reach them. The construction of routing table is the primary goal of routing protocols.

RTT : Acronym for Round Trip Time. It is the elapsed time for a message to transit between two nodes of a network.

Timeout : A specified period of time that will be allowed to elapse in a system before a specified event is to take place, unless another specified event occurs first; in either case, the period is terminated when either event takes place.

VO : Acronym for Virtual Organization. Set of users which wants to share their own resources to the organization they belong in order to achieve enough computing resources to carry out their common objectives.

WSRF : Acronym for Web Services Resource Framework. A web service by itself is nominally stateless, i.e., it retains no data between invocations. This limits the things that can be done with web services. WSRF provides a set of operations that web services may implement to become stateful.

Bibliography

[1] I. Foster, "What is the grid: a three point check-list," tech. rep., 2002.

[2] I. Foster, C. Kesselman, and S. Tuecke, "Anatomy of the grid: Enabling scalable virtual organizations, the," *International Journal on High Performance Computing Applications*, pp. 200–222, 2001.

[3] Homepage of the Grid4All Consortium. WebSite, June 2006. `http://www.grid4all.eu`.

[4] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, "Sharp: an architecture for secure resource peering," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 133–148, ACM Press, 2003.

[5] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum, "Sharing network resources with brokered leases," in *USENIX Technical Conference*, June 2006.

[6] D. Abrazhevich, "Classification and characteristics of electronic payment systems," *Lecture Notes in Computer Science*, pp. 81–90, 2001.

[7] K. Lai and L. Rasmusson, "Tycoon: an implementation of a distributed, market-based resource allocation system," tech. rep., November 2005.

[8] D. Hausheer and B. Stiller, "Peermint: Decentralized and secure accounting for peer-to-peer applications.," in *NETWORKING*, pp. 40–52, 2005.

[9] D. Hausheer and B. Stiller, "Peermart: the technology for a distributed auction-based market for peer-to-peer services," in *IEEE International Conference*, pp. 1583–1587, May 2005.

[10] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat, "Resource allocation in federated distributed computing infrastructures," in *1st Workshop on Operating System and Architectural Support for the Ondemand IT Infrastructure*, October 2004.

[11] B. Yang and H. Garcia-Molina, "Ppay: Micropayments for peer-to-peer systems," tech. rep., Stanford University, 2003. http://dbpubs.stanford.edu/pub/2003-31.

[12] V. Vishnumurthy, S. Chandrakumar, and E. Sirer, "Karma : A secure economic framework for peer-to-peer resource sharing," in *Proceedings of the Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[13] F. D. Garcia and J.-H. Hoepman, "Off-line karma: A decentralized currency for peer-to-peer and grid applications," November 2004.

[14] O. Regev and N. Nisan, "Popcorn market: an online market for computational resources, the," in *Proceedings of the First International Conference on Information and Computation Economies*, (New York, NY, USA), pp. 148—157, ACM Press, 1998.

[15] A. Barmouta and R. Buyya, "Gridbank: a grid accounting services architecture (gasa) for distributed systems sharing and integration," in *Parallel and Distributed Processing SymposiumParallel and Distributed Processing Symposium*, p. 8, April 2003.

[16] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," in *Proceedings of the 8th Annual International Cryptology Conference on Advances in Crytology*, (London, UK), pp. 319–327, Springer-Verlang, 1990.

[17] S. Brands, "Untraceable off-line cash in wallet with observers (extended abstract)," in *Advances in Cryptology – CRYPTO '93* (D. R. Stinson, ed.), vol. 773 of *Lecture Notes in Computer Science*, pp. 302–318, Springer-Verlag, 22–26 August 1993.

[18] T. Okamoto and K. Ohta, "Universal electronic cash," in *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, (London, UK), pp. 324–337, Springer-Verlag, 1992.

[19] Y. Mu, K. Q. Nguyen, and V. Varadharajan, "A fair electronic cash scheme," in *ISEC '01: Proceedings of the Second International Symposium on Topics in Electronic Commerce*, (London, UK), pp. 20–32, Springer-Verlag, 2001.

[20] Z. Jia, S. Tiange, H. Liansheng, and D. Yiqi, "A new micro-payment protocol based on p2p networks," in *E-Business Engineering, 2005. ICEBE 2005. IEEE International Conference*, October 2005.

[21] N. C. Liebau, V. Darlagiannis, A. Mauthe, and R. Steinmetz, "A token-based accounting scheme for p2p-systems," tech. rep., May 2004.

[22] M. Sirbu and J. Tygar, "Netbill: An internet commerce system optimized for network delivered services," *CompCon*, vol. 0, p. 20, 1995.

[23] B. Cox, J. Tygar, and M. Sirbu, "Netbill security and transaction protocol," in *First USENIX Workshop on Electronic Commerce, The*, pp. 77–88, July 1995.

[24] R. L. Rivest and A. Shamir, "Payword and micromint: Two simple micropayment schemes," in *Security Protocols Workshop*, pp. 69–87, 1996.

[25] R. L. Rivest, "Electronic lottery tickets as micropayments," in *Financial Cryptography* (R. Hirschfeld, ed.), (Anguilla, British West Indies), pp. 307–314, Springer, 1997.

[26] S. Micali and R. L. Rivest, "Micropayments revisited," in *CT-RSA*, pp. 149–163, 2002.

[27] R. L. Rivest, "Peppercoin micropayments," in *Financial Cryptography* (A. Juels, ed.), vol. 3110, pp. 2–8, 2004.

[28] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner, "Ikp – a family of secure electronic payment protocols," pp. 89–106.

[29] R. Hunt, "Pki and digital certification infraestructure," in *Proceedings. Ninth IEEE International Conference on Networks*, pp. 234–239, 2001.

[30] A. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[31] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *STOC '97: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pp. 654–663, 1997.

[32] C. G. P. Richa, R. Rajaraman, and A. W., "Accessing nearby copies of replicated objects in a distributed environment," in *ACM Symposium on Parallel Algorithms and Architectures*, pp. 311–320, 1997.

[33] F. Dabek, B. Zhao, P. Druschel, and J. Kubiatowicz, "Towards a common api for structured peer-to-peer overlays," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, February 2003.

[34] S. El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*. PhD thesis, Stockholm, Sweden, June 2005.

[35] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer Vol.30, Issue 4*, pp. 68–74, April 1997.

[36] P. Druschel and A. Rowstron, "Past: a large-scale, persistent peer-to-peer storage utility," in *Proceedings of the Eigth Workshop on Hot Topics in Operating Systems*, pp. 75–80, May 2001.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 149–160, 2001.

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

[39] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 161–172, 2001.

[40] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infraestructure for fault-tolerant wide-area location and routing," tech. rep., April 2001.

[41] A. Ghodsi, *Distributed K-Ary System: Algorithms for Distributed Hash Tables*. PhD thesis, Stockholm, Sweden, December 2006.

[42] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer network schemes," *Communications Surveys and Tutorials, IEEE*, March 2004.

[43] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, "Dks(n, k, f): a family of low communication, scalable and fault-tolerant infrastructures for p2p applications," in *Cluster Computing and the Grid*, pp. 344–350, 2003.

[44] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public dht service and its uses," in *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 73–84, 2005.

[45] L. Lamport, "Paxos made simple," *SIGACT News*, pp. 18–25, 2001.

[46] M. Raynal and M. Singhal, "Logical time: A way to capture causality in distributed systems," tech. rep., 1995.

[47] S. Sankararaman, B.-G. Chun, C. Yatin, and S. Shenker, "Key consistency in dhts," tech. rep., November 2005.

[48] A. Muthitacharoen, S. Gilbert, and R. Morris, "Etna: a fault-tolerant algorithm for atomic mutable dht data," tech. rep., June 2005.

[49] B. Temkow, A. M. Bosneag, X. Li, and M. Brockmeyer, "Paxondht: Achieving consensus in distributed hash tables," in *International Symposium on Applications and the Internet*, Jan 2006.

[50] R. Akbarinia and V. Martins, "Data management in the appa p2p system," in *International Workshop on High-Performance Data Management in Grid Environments*, 2006.

[51] M. Velazquez, "A survey of distributed mutual exclusion algorithms," tech. rep., 1993.

[52] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[53] M. Raynal, "A simple taxonomy for distributed mutual exclusion algorithms," *SIGOPS Operating Systems Rev.*, vol. 25, pp. 47–50, 1991.

[54] W. Chen, S. Lin, Q. Lian, and Z. Zhang, "Sigma: a fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses," in *Proceedings. 11th Pacific Rim International Symposium on Dependable Computing*, p. 8, December 2005.

[55] M. Muhammad, A. S. Cheema, and I. Gupta, "Efficient mutual exclusion in peer-to-peer systems," 2005.

[56] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *2nd International Conference on Distributed Computing Systems*, p. 12, 1981.

[57] Foster, I., K. Czajkowski, D. E. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke, "Modeling and managing state in distributed systems: the role of ogsi and wsrf," in *Proceedings of the IEEE, Vol.93, Iss. 3*, pp. 604–612, March 2005.

[58] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. B. Stefani, "Fractal component model and its support in java, the," *Software - Practice and Experience*, pp. 1257 – 84, October 2006.

[59] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, *Programming, Deploying, Composing for the Grid*. Springer-Verlag, January 2006.

[60] A. Ouorou, E. Gourdin, N. Amara, R. Krishnaswamy, L. Navarro, R. Brunner, X. León, and X. Vilajosana, "Requirements for market-based resource management and state of the art," in *Public Deliverable 2.1 (Month 12) of Grid4All*, June 2007.

[61] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *Computer Communication Review*, pp. 3–12, July 2003.

[62] AETIC (Asociación Española de Electrónica, Tecnologías de la Información y Telecomunicaciones), January 2006. `http://www.aetic.es`.

[63] Grid Market Middleware, 2007 `http://recerca.ac.upc.edu/gmm/`.

[64] OneJar: an application to deliver an application in only one jar regardless the jar dependencies `http://one-jar.sourceforge.net/`.

[65] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," *VLDB Journal: Very Large Data Bases*, vol. 5, no. 1, pp. 48–63, 1996.

[66] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi, "Self-recharging virtual currency," in *P2PECON '05: Proceeding of the 2005 ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems*, (New York, NY, USA), pp. 93–98, ACM Press, 2005.

[67] R. Anderson, C. Manifavas, and C. Sutherland, "Netcard – a practical electronic cash system."

[68] E. Turcan and R. L. Graham, "Getting the most from accountability in p2p," in *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, 2001.

[69] L. Lamport, R. Shostak, and M. Pease, "Byzantine general problem, the," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[70] T. Poutanen, H. Hinton, and M. Stumm, "Netcents: A lightweight protocol for secure micropayments," pp. 25–36.

[71] J. Su and J. Tygar, "Building blocks for atomicity in electronic commerce," in *Proceedings of USENIX Security Symposium*, 1996.

[72] M. Manasse, S. Glassman, M. Abadi, P. Gauthier, and P. Sobalvarro, "Millicent protocol for inexpensive electronic commerce, the."

[73] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[74] D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures," in *International Association for Cryptologic Research, The*, vol. 13, pp. 361–396, 2000.

[75] D. Chaum and T. Pryds, "Transferred cash grows in size," in *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, p. 390, May 1992.

[76] N. A. Lynch, D. Malkhi, and D. Ratajczak, "Atomic data access in distributed hash tables," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pp. 295–305, 2002.

[77] P. Knezevic, A. Wombacher, and T. Risse, "Highly available dhts: Keeping data consistency after updates," in *4th International Workshop, AP2PC 2005, July 25, 2005, Revised Papers, Utrecht, Netherlands*, pp. 70–80, July 2006.

[78] A. Ghodsi, L. O. Alima, and S. Haridi, "Symmetric replication for structured peer-to-peer systems," in *Proceedings of The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, p. 12, 2005.

[79] V. Mesaros, R. Collet, K. Glynn, and P. Van Roy, "A transactional system for structured overlay networks," tech. rep., March 2005.

[80] L. Shi-Ding, Q. Lian, M. Chen, and Z. Zhang, "A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems," *IPTPS 2004 : International Workshop on Peer-to-Peer Systems*, pp. 11–21, February 2004.

[81] H. Kopetz and P. Verissimo, "Real time and dependability concepts," *Distributed Systems (2nd Ed.)*, pp. 411–446, 1993.

[82] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., 1994.

[83] D. Powell, "Group communication," *Communications of the ACM*, vol. 39, pp. 50–53, 1996.

[84] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured p2p networks," in *2nd International Workshop On Peer-To-Peer Systems (IPTPS'03), The*, February 2003.

[85] A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, "Self-correcting broadcast in distributed hash tables," in *Series on Parallel and Distributed Computing and Systems (PDCS'2003)*, 2003.

[86] A. Ghodsi, L. O. Alima, and S. Haridi, "Low-bandwidth topology maintenance for robustness in structured overlay networks," in *38th International HICSS Conference, The*, January 2005.