Project no. 034567

Grid4All

Specific Targeted Research Project (STREP)

Thematic Priority 2: Information Society Technologies

Technical Annex (D2.3)

Currency Management System design and implementation details

Start date of project: 1 June 2006 Duration: 30 months

Author: Xavier León, Universitat Politècnica de Catalunya (UPC)

Revision 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Technical details of the *Currency Management Service*

Xavier León
xleon@ac.upc.edu

Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)

June 26, 2008

Contents

List of Figures

List of Tables

Chapter 1

General Overview

This chapter defines the currency management used within Grid4All. It is a simple introduction to the idea on top of which the CMS (and therefore the prototype which is 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 in previous related work.

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 un-

derstood 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 2

System specification

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

2.0.1 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 concret 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 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.

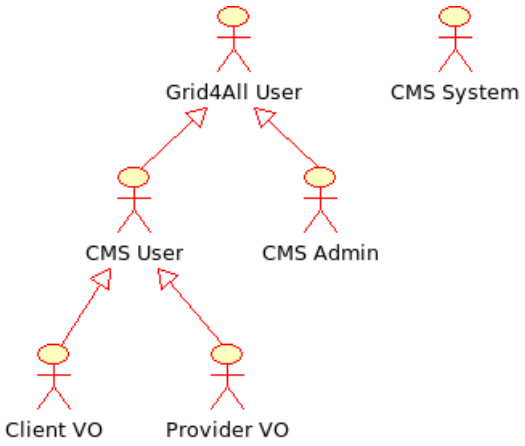


Figure 1: Both Client and Provider are Grid4All users. Client will be the responsible to initiate the payment mechanism by transfering funds to its provider

2.0.2 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.

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 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.

2.1 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).

Chapter 3

System Architecture

Throughout this chapter, 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.

3.0.1 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.

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 1).

3.0.2 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 2 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

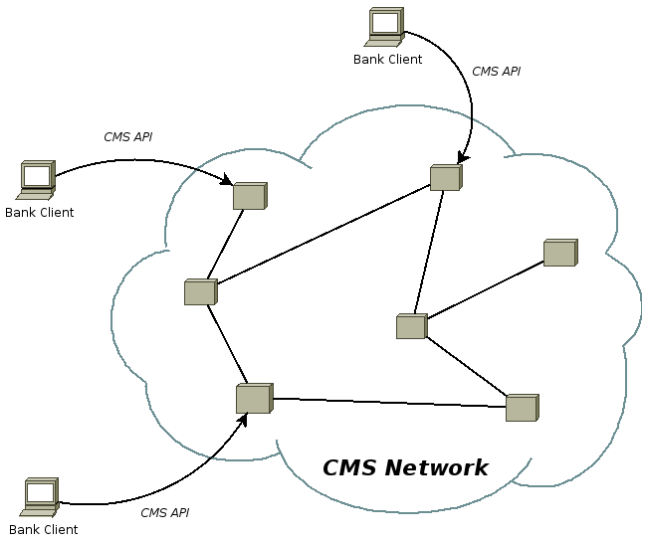


Figure 1: Currency Management System deployment view. The CMS Network is constructed upon some nodes each of which is running the prototype of the CMS. Clients access the CMS functionalities through any of the nodes through the external API.

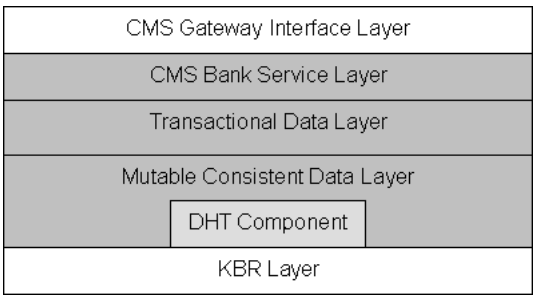


Figure 2: Currency Management System Component Architecture. The CMS follows a layered architecture isolating different responsabilities in each layer.

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 external 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 , Fractal 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. 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. 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. 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 4 presents a concrete view on the protocols and design decisions to achieve the responsabilities of each layer.

3.0.3 Roles

The last issue regarding architecture is the roles which can be exerted by each one of the nodes. 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 4

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 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 4.1, 4.2, 4.3 and 4.4. Implementation details of common components or specific features will be introduced in Section 4.5.

4.1 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 synchronous 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*. 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.

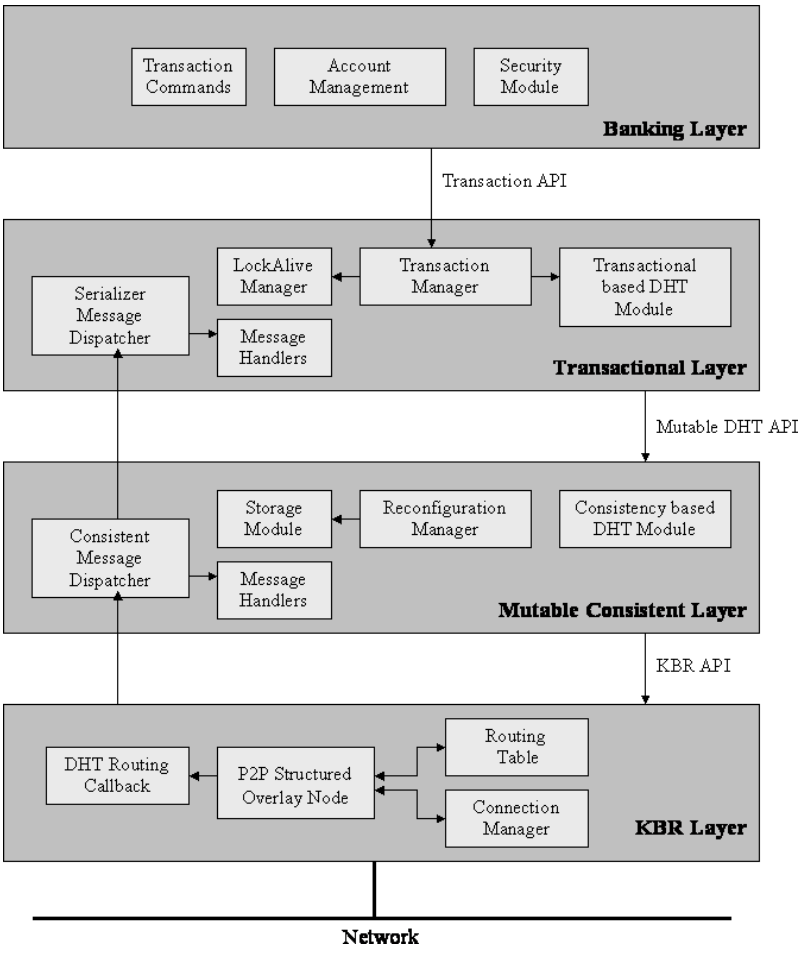


Figure 1: Currency Management System layered view.

4.1.1 Key Based Routing API

Figure 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.

4.1.1.1 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.

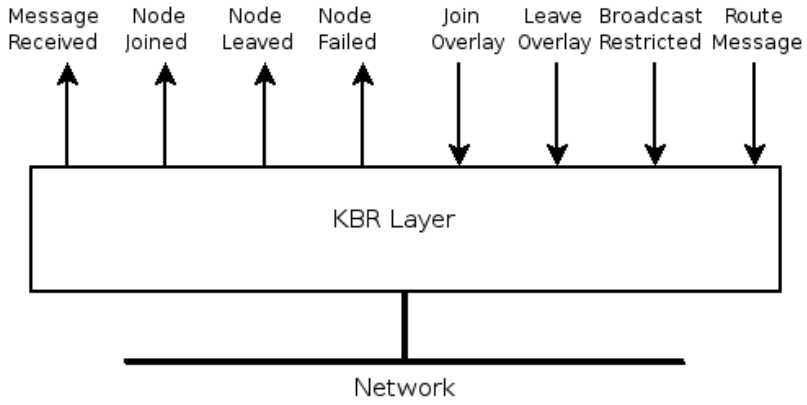


Figure 2: KBR Component Interface: lines to the upper layer are notifications of events in which the Mutable Consistent Layer will be interested in order to reconfigure its state. The basic communication primitives used are route a message, and broadcast a message to a set of nodes within the range introduced.

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

4.1.1.2 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.

4.2 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 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.

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*.

4.2.1 Mutable Consistent DHT API

A first scheme of the API is shown in Figure 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.

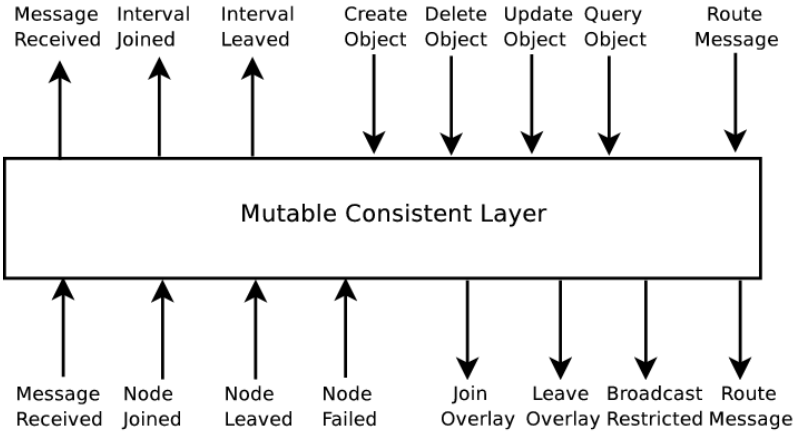


Figure 3: Mutable Consistent Component Interface: events sent to the upper layer are intervalJoin, intervalLeave and message Arrived. Moreover, it offers basic enhanced DHT operations in order to support consistent updating of objects.

[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

4.2.1.1 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.

4.2.1.2 External Invocations

This is the first layer implemented by us to support the concrete necessities of the upper layers. In Section 4.5, 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 4.5. 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 4.2.3, we provide a simple API representing a DHT. We have enhanced the common API described in **?** 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.

4.2.2 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).

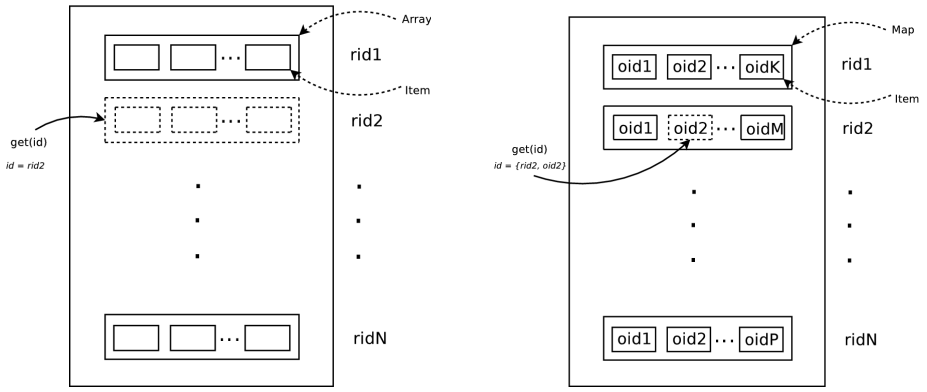
To achieve this purpose, we define a 2-dimensional identifier space where the first coordinate is the current DHT identifier space and the second coordinate is the position within the first coordinate. In other words, we assign each object a *Mutable Identifier* which is constructed by two identifiers:

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 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



(a) DKS Identifier assignment

(b) Mutable Identifier assignment

Figure 4: Figure 4(a) shows how DKS manages the get operation taking into account its own identifier assignment. Figure 4(b) shows how our Mutable DHT Layer handles the identifier space enabling the retrieving of a single object instead of a set.

store always under the same node the object stored and its associated state, enabling the efficient retrieving of items.

4.2.3 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 **?**, 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 serial-

izing 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 symmetric replication approach, the replicated identifiers are defined as follows:

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

where i is the identifier, N is the identifier space, f is the replication degree and x is the x^{th} replica within the range [1,f]. 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 4.1, 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 *mutableObject* associated to the object, its timestamp is set to zero to start the increasing numbering.

As shown in Algorithm 4.2, 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

Algorithm 4.1: 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
```

Algorithm 4.2: 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
```

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.

Algorithm 4.3: 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 4.3 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 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.

4.2.4 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 **?** but differs substantially when dealing with failures.

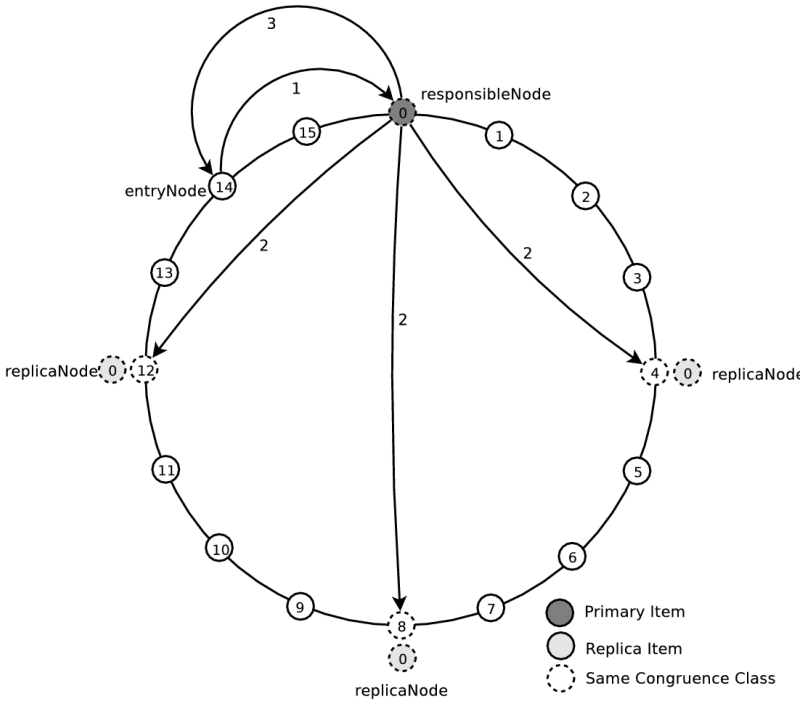


Figure 5: Basic Mutable Layer Communication Protocol. Assume a fully populated ring with an identifier space N set to 16 and replication factor f set to 4. Step one: the *entryNode* sends the corresponding message to the *responsibleNode*. Step two: the *responsibleNode* process the message and perform 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 4.4 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 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.

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.

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

Algorithm 4.4: Interval Reconfiguration Mechanism to deal with node failures

```
1: procedure DHT.intervalFailed (Interval failed)
2:     for all x in 1 ≤ x ≤ 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 ≤ 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
```

4.3 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 **??**, 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 **??** 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.

4.3.1 Transactional Layer API

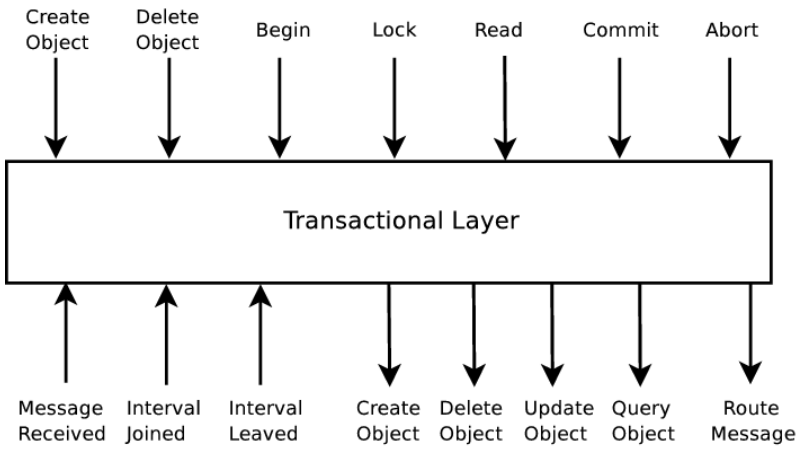


Figure 6: Transactional Component Interface: this layer does not provide any information about the underlying overlay network. This way, it abstracts completely the management complexity by providing a powerfull while simple Transaction API.

4.3.1.1 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.

4.3.1.2 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 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 4.3.2), 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.

4.3.2 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 **??**(Evaluation) which load degree the coordinator is able to manage.

4.3.2.1 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 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).

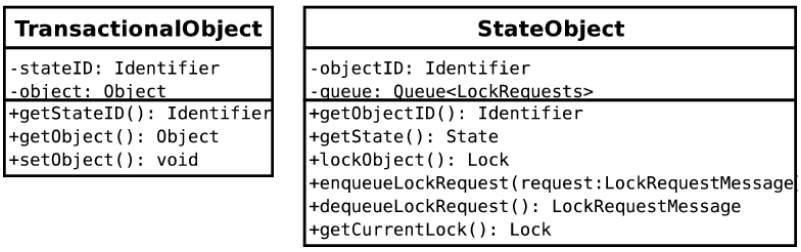


Figure 7: Two different objects stored within the Mutable Consisten DHT Layer. The TransactionalObject stores the identifier of its associated state. The StateObject stores the identifier of its associated object and provides some operations to manage it.

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.

4.3.2.2 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 4.5 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 4.5: 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 4.6, 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 4.6: 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.

4.3.2.3 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 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 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*).

Algorithm 4.7: 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
```

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 4.6. 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 4.2, 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.

4.3.2.4 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 **??**, 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* **?**. 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.

4.3.3 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.

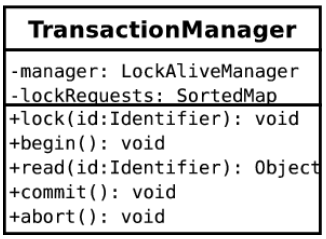


Figure 8: TransactionManager Class Diagram

Basically, as shown in Figure 8, what we offer are operations to create, delete and lock objects as well as begin, commit and abort transactions. As explained in Section 4.3.1, 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 pre-stablished 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 4.8 shows an skeleton of a transaction, based on which every transaction can be implemented.

Algorithm 4.8: 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
```

4.4 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*.

4.4.1 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 9 shows different commands implemented in the Banking Layer in order to fulfill the API presented in Table 1 (CMS Banking Layer API).

Notice that the *Command* abstract class has two implemented methods to store and retrieve the result of transaction. This result will always be of abstract type *Receipt* to ease the handle of different result type.

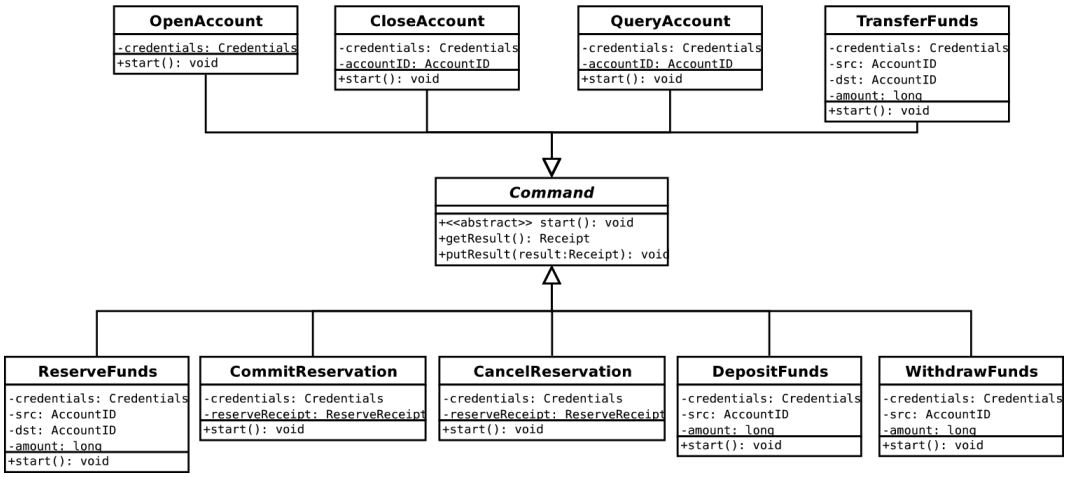


Figure 9: CMS Banking Layer Commands

4.4.2 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 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).

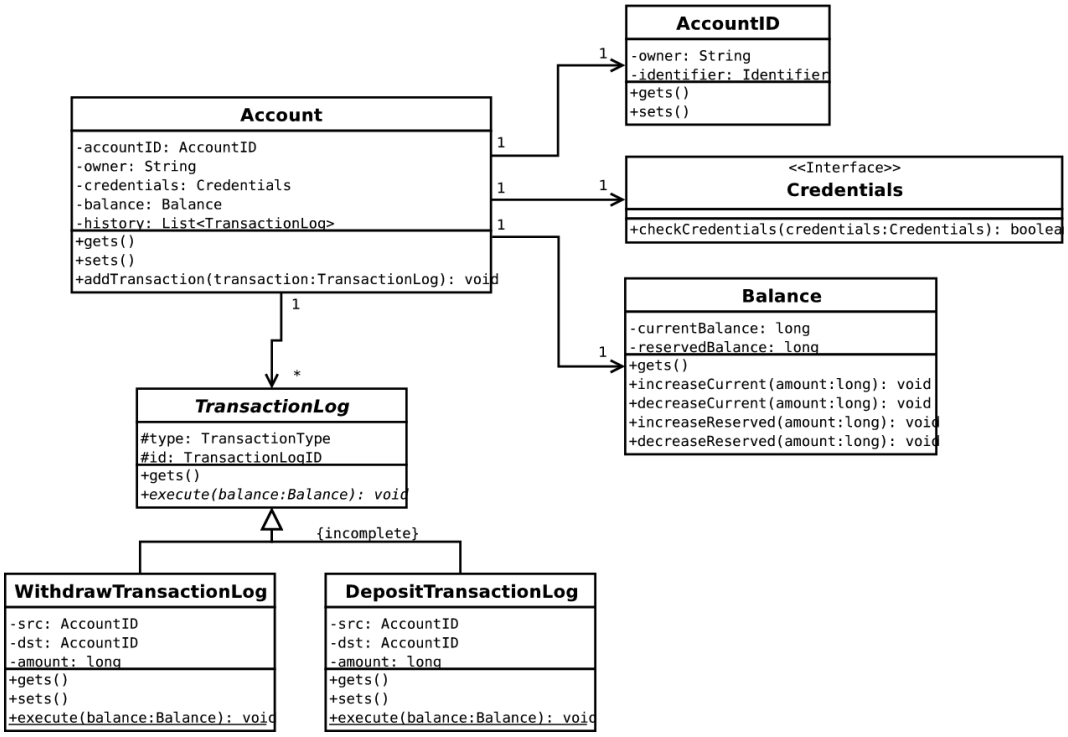


Figure 10: Account and Receipts Class Diagram

4.4.3 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 **??**.

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 11, 12 and 13). The rest of transactions follow the same scheme changing only the error handling and the account modifications made.

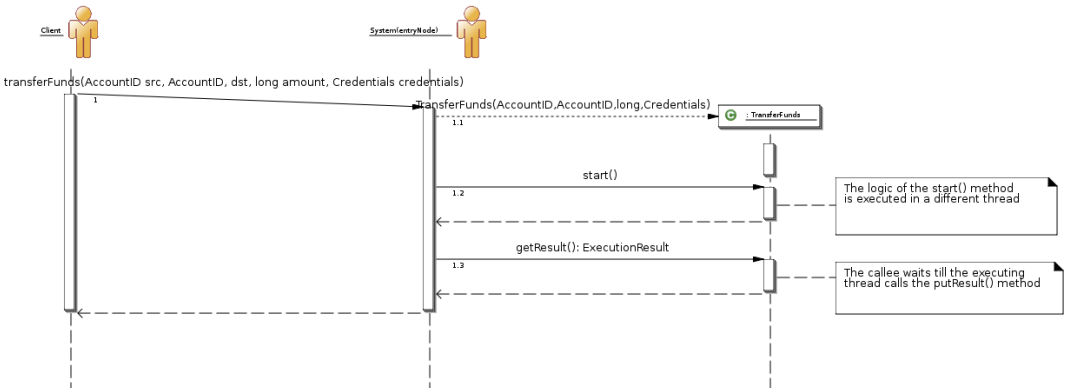


Figure 11: Example of CMS API call through an abstract gateway node. The client asks through a previously known mechanism (Web-Service, WSRF, Internal Protocol, etc.) to a concrete API method. It shows how the entry node creates the Command object and execute it in a separate thread.

4.5 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.

4.5.1 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 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 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 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. 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 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 **??**, this mechanism introduces a very little overhead when dealing with high concurrent load, due to a small necessary synchronized part of code .

4.5.2 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 18 and Figure 19 shows this mechanism.

Figure 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.

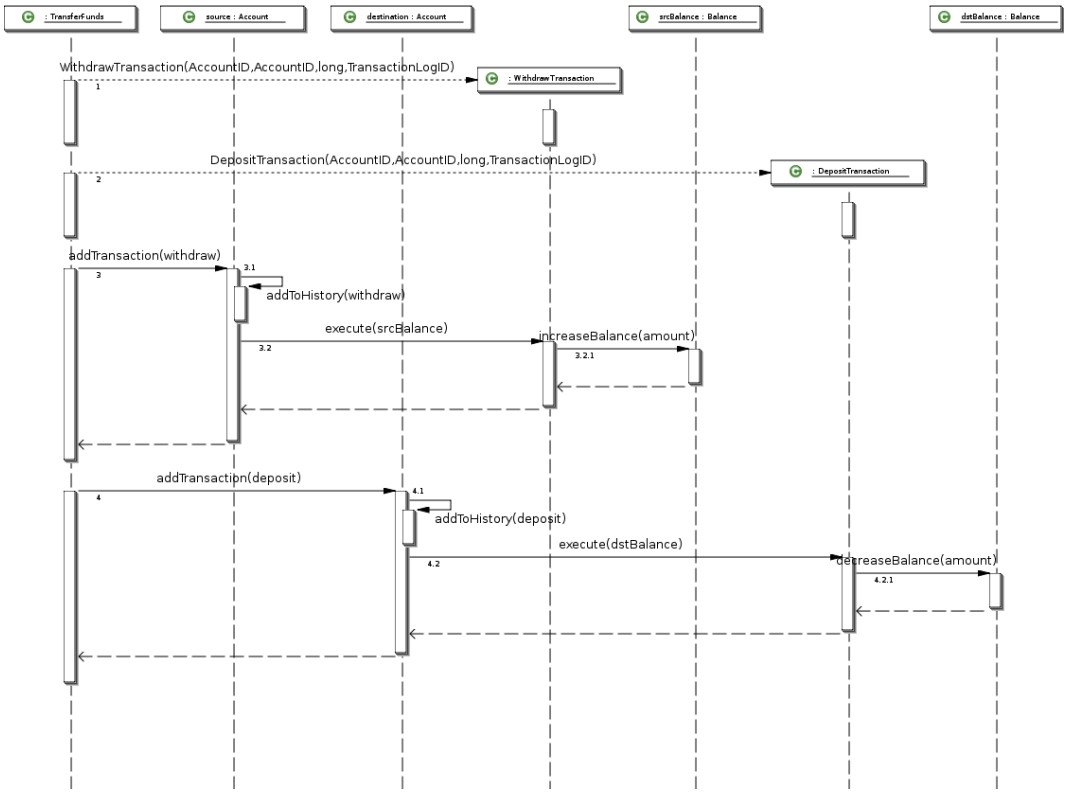


Figure 13: Example of the logic of the TransferFunds transaction. It shows how both deposit and withdraw transaction logs are created for both account modifications. Once created, each transaction log is inserted within the account which means: a) add this transaction to the history of transactions and, b) execute the transaction which is to increase or decrease the balance of the account depending if the transaction is deposit or withdraw respectively.

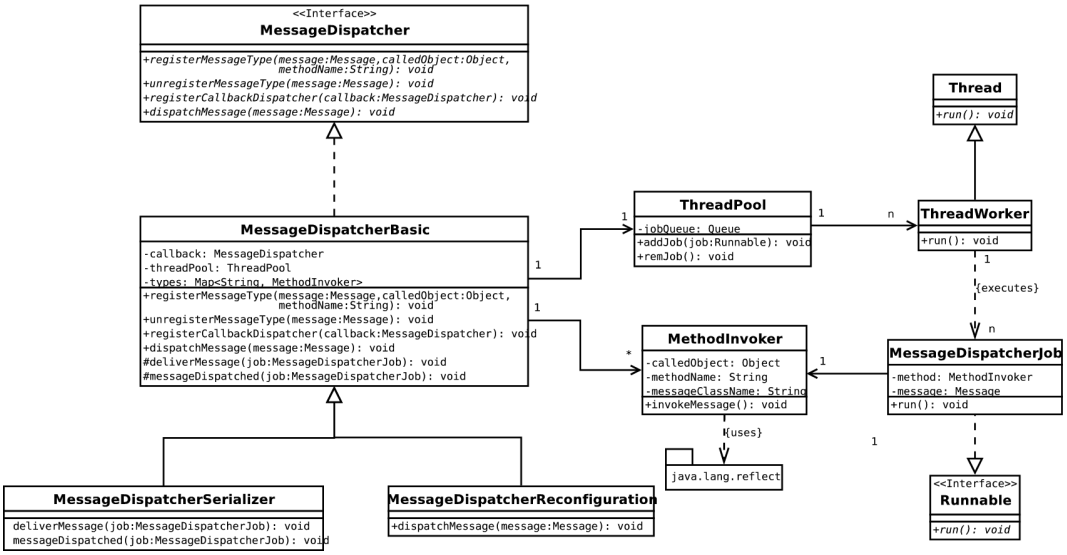


Figure 14: Message Dispatcher Class Diagram

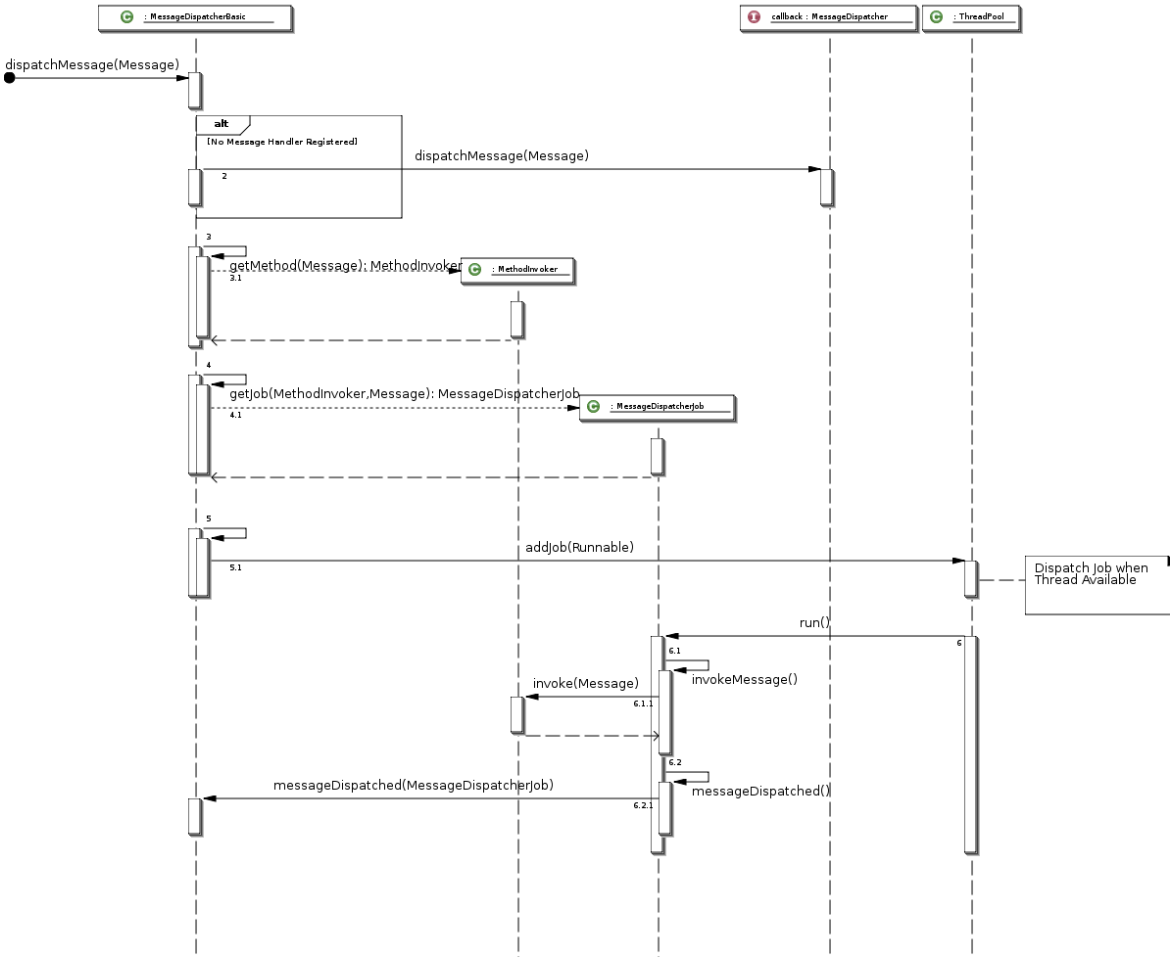


Figure 15: Message Dispatcher Sequence Diagram

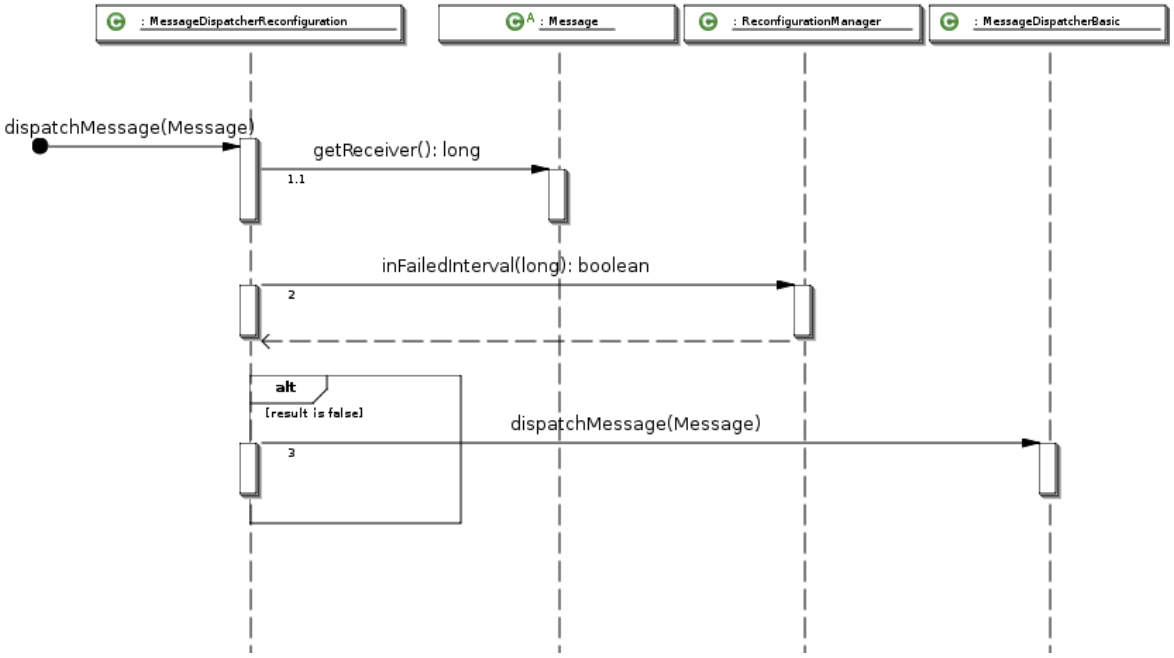


Figure 16: Message Dispatcher Reconfiguration Sequence Diagram. This diagram shows the reimplementation of the dispatchMessage method inherited from the MessageDispatcherBasic class. If the destination identifier is within a failed interval, the message is discarded to avoid inconsistent responses.

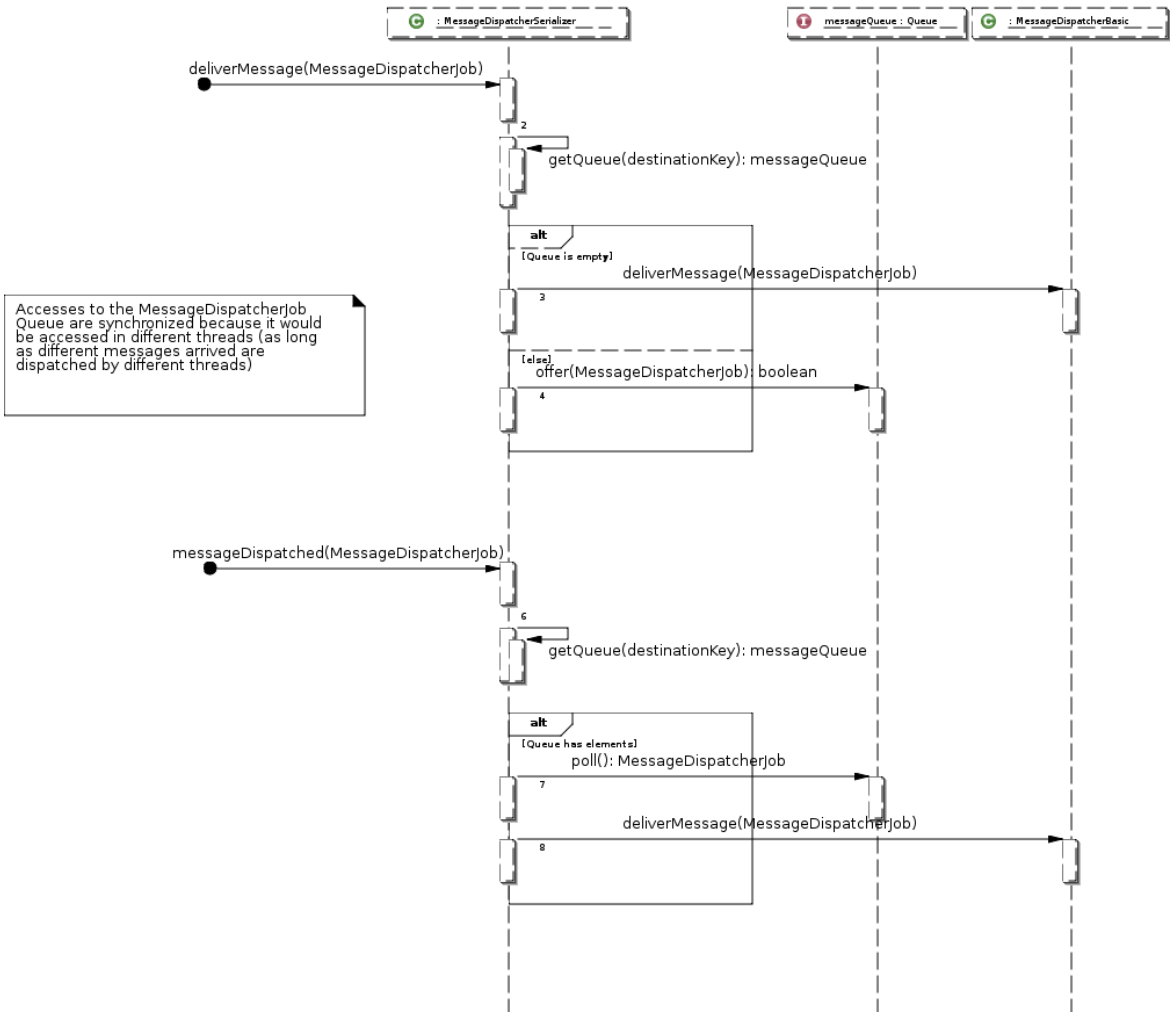


Figure 17: Message Dispatcher Serializer Sequence Diagram. Reimplementation of two methods provided by the basic message dispatcher. Both deliverMessage and messageDispatcher are different methods although they are presented within the same sequence diagram.

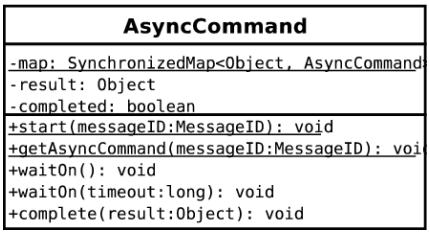


Figure 18: Synchronous Communication Class Diagram. This class provides two basic static operations (underlined) to start and get a previously created AsyncCommand Instance. The waitOn() and waitOn(long timeout) are operations to wait till the response is received with and without an spceified timeout). The complete method is used upon the reception of a message to wakeup a thread waiting on this object.

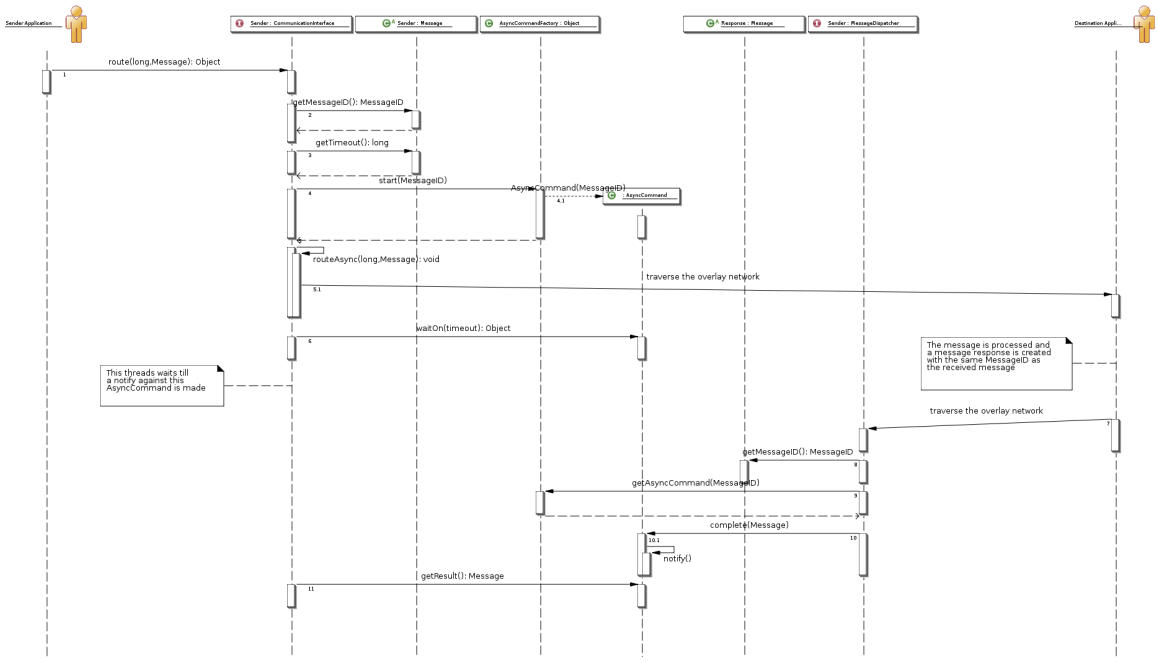


Figure 19: Synchronous Communication Sequence Diagram. This diagram shows how the thread responsible to send the message waits till the response arrives. Thereafter, the thread is woken up and is able to continue its execution taking into account the response.

Bibliography

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.

A. Ghodsi, *Distributed K-Ary System: Algorithms for Distributed Hash Tables*. PhD thesis, Stockholm, Sweden, December 2006.

R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer Vol.30, Issue 4*, pp. 68–74, April 1997.

N. C. Liebau, V. Darlagiannis, A. Mauthe, and R. Steinmetz, "A token-based accounting scheme for p2p-systems," tech. rep., May 2004.

D. Hausheer and B. Stiller, "Peermint: Decentralized and secure accounting for peer-to-peer applications.," in *NETWORKING*, pp. 40–52, 2005.

F. D. Garcia and J.-H. Hoepman, "Off-line karma: A decentralized currency for peer-to-peer and grid applications," November 2004.

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.

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.

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.

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.

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.

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.

R. L. Rivest and A. Shamir, "Payword and micromint: Two simple micropayment schemes," in *Security Protocols Workshop*, pp. 69–87, 1996.

R. Anderson, C. Manifavas, and C. Sutherland, "Netcard – a practical electronic cash system."

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.

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.

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.

T. Poutanen, H. Hinton, and M. Stumm, "Netcents: A lightweight protocol for secure micropayments," pp. 25–36.

J. Su and J. Tygar, "Building blocks for atomicity in electronic commerce," in *Proceedings of USENIX Security Symposium*, 1996.

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.

M. Sirbu and J. Tygar, "Netbill: An internet commerce system optimized for network delivered services," *CompCon*, vol. 0, p. 20, 1995.

M. Manasse, S. Glassman, M. Abadi, P. Gauthier, and P. Sobalvarro, "Millicent protocol for inexpensive electronic commerce, the."

S. Micali and R. L. Rivest, "Micropayments revisited," in *CT-RSA*, pp. 149–163, 2002.

R. L. Rivest, "Electronic lottery tickets as micropayments," in *Financial Cryptography* (R. Hirschfeld, ed.), (Anguilla, British West Indies), pp. 307–314, Springer, 1997.

J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

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.

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.

D. Abrazhevich, "Classification and characteristics of electronic payment systems," *Lecture Notes in Computer Science*, pp. 81–90, 2001.

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.

K. Lai and L. Rasmusson, "Tycoon: an implementation of a distributed, market-based resource allocation system," tech. rep., November 2005.

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.

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.

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.

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.

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.

R. L. Rivest, "Peppercoin micropayments," in *Financial Cryptography* (A. Juels, ed.), vol. 3110, pp. 2–8, 2004.

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.

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.

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.

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.

R. Akbarinia and V. Martins, "Data management in the appa p2p system," in *International Workshop on High-Performance Data Management in Grid Environments*, 2006.

A. Muthitacharoen, S. Gilbert, and R. Morris, "Etna: a fault-tolerant algorithm for atomic mutable dht data," tech. rep., June 2005.

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.

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.

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.

S. Sankararaman, B.-G. Chun, C. Yatin, and S. Shenker, "Key consistency in dhts," tech. rep., November 2005.

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.

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.

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.

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.

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.

V. Mesaros, R. Collet, K. Glynn, and P. Van Roy, "A transactional system for structured overlay networks," tech. rep., March 2005.

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.

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.

M. Muhammad, A. S. Cheema, and I. Gupta, "Efficient mutual exclusion in peer-to-peer systems," 2005.

A. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

H. Kopetz and P. Verissimo, "Real time and dependability concepts," *Distributed Systems (2nd Ed.)*, pp. 411–446, 1993.

V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., 1994.

D. Powell, "Group communication," *Communications of the ACM*, vol. 39, pp. 50–53, 1996.

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.

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.

R. Hunt, "Pki and digital certification infraestructure," in *Proceedings. Ninth IEEE International Conference on Networks*, pp. 234–239, 2001.

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.

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.

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.

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.

B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infraestructure for fault-tolerant wide-area location and routing," tech. rep., April 2001.

S. El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*. PhD thesis, Stockholm, Sweden, June 2005.

Homepage of the Grid4All Consortium. WebSite, June 2006. `http://www.grid4all.eu`.

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.

L. Lamport, "Paxos made simple," *SIGACT News*, pp. 18–25, 2001.

M. Raynal and M. Singhal, "Logical time: A way to capture causality in distributed systems," tech. rep., 1995.

M. Velazquez, "A survey of distributed mutual exclusion algorithms," tech. rep., 1993.

N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

M. Raynal, "A simple taxonomy for distributed mutual exclusion algorithms," *SIGOPS Operating Systems Rev.*, vol. 25, pp. 47–50, 1991.

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.

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.

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.

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.

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.

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.

AETIC (Asociación Española de Electrónica, Tecnologías de la Información y Telecomunicaciones), January 2006. `http://www.aetic.es`.

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.

I. Foster, "What is the grid: a three point check-list," tech. rep., 2002.

OneJar: an application to deliver an application in only one jar regardless the jar dependencies `http://one-jar.sourceforge.net/`.

Grid Market Middleware, 2007 `http://recerca.ac.upc.edu/gmm/`.