# URL-Shortener

Francesco Balzano

April 26, 2017

# Contents

# 1 Overview

URL shortening is the translation of a long Uniform Resource Locator (URL) into an abbreviated alternative that redirects to the longer URL. A shortened URL may be desired for messaging technologies that limit the number of characters in a message, for reducing the amount of typing required if the reader is copying a URL from a print source, for making it easier for a person to remember, or for the intention of a permalink.
The aim of this project is to provide the implementation of a distributed url-shortener service.

# 2 Design choices

There are three fundamental metrics in the assessment of a distributed system: Consistency, Availability and Partition Tolerance. It is well known from the CAP theorem that it is impossibile to achieve all of them at the same time in a distributed system.
In this project, I have chosen to focus on availability and partition tolerance, at the expense of consistency. In particular, the consistency model is not a strong one. In the following subsections, I list and explain the design choices that I made.

## 2.1 API

The url-shortener project is a service that provides the following APIs:

**get** Given the shorten url returns the original url, if present.
`get(shortUrl) -> longUrl`

**put** Generates and returns the shortened url associated with the provided url.
`put(longUrl) -> shortUrl`

**remove** Removes the couple `<shortened url, original url>`.
`remove(shortUrl) -> longUrl`

## 2.2 Passive Replication

Clients can communicate with every node. If the contacted node is the primary for that request, it will execute the request and directly return the result to the client; otherwise it will forward the request to the primary, which will execute it and send back the response to the sending node, which in turn will hand it back to the client.

I have chosen the Passive Replication strategy because I think it should keep lower the number of version conflicts.

## 2.3 Data Partitioning

The partitioning strategy to map objects into nodes is a dynamic one: namely, it is employed Consistent Hashing with virtual nodes. The use of virtual nodes has potentially two advantages: in case of heterogeneous machines we can assign more virtual nodes to the most powerful machines, so that it will be more likely that these machines will handle a bigger number of items. The other advantage concerns homogeneous machines (`i.e.` machines with similar computational power, bandwidth, storage...). In this case the use of virtual nodes results in a more fair distribution of items among nodes with higher probability.

In case of node leave, due for instance to node crash or network partition, the keys asssigned to this node are automatically assigned to another, working node. Since we have replication of data and we use the gossip protocol to detect dead nodes, we are capable to face the leave of one or more nodes without compromising the functioning of the whole system. In other words, availability and partition tolerance are guaranteed.

## 2.4 Data Replication

We want to achieve availability, so an asynchronous replication strategy is adopted. Namely, the primary node immediately answers to the client after an operation is performed. Messages to the backup nodes are sent periodically, and not after every update of the primary's data. This strategy reduces the latency of the communication client-primary, again at the expense of consistency.
Each primary node is associated 1 backup node, which is the next node clockwise in the ring. Every time a primary wishes to update its replica, it sends all the content of its database to this node.

## 2.5 Primary Failure

The use of Consistent Hashing and the the specific data replication strategy adopted (backup is the next node clockwise in the ring) allows to have a vry easy failover procedure: if node $i$ is down, the space of keys that belonged to node $i$ will be automatically mapped to node *(i+1) mod n*. Since node *(i+1) mod n* is the backup of node $i$ it will have a copy of the keys of node $i$ (although possibly not updated), so the correct behaviour will be maintained also in case of failure of the primary without needing an explicit failover procedure. For further details about the failover procedure please refer to the *Implementation* section.

## 2.6 Consistency Model

This project adopts an eventual consistency model. It is possible to get different results if running the same query at the same time on the leader and on a follower. This may happen if the follower has an outdated version of the leader's data. Anyway this is only a temporary state: if we stop writing to the database for a while, the followers will eventually become consistent with the leader. I decided to accept this weak consistency model in order to have better availability and partition tolerance.

In realt tutte le richieste arrivano a un leader

## 2.7 Conflicts Resolution

Each node is assigned a vector clock that allows it to order events, that is to track causal dependencies between events. When a message is sent from a node to another, it carries both the object (the target of the communication) and the sender's vector clock. If the recipient node already has that object with the associated vector clock in its database, two situations may arise:

- If one vector clock is smaller than the other, keep the greatest (i.e. latest) version of the object;
- If that is not the case, the two vector clocks are concurrent, and if the two objects are assigned different values then we have a conflict.

Non ti puoi affidare al client per risolvere inconsistenze, va fatto server-side

# 3 Implementation

There are clients, that invoke the service, and nodes, that comprise the cluster and actually implements the service. Clients and nodes stay in two different modules: the formers in the client module, the latters in the core module. Indeed they are logically different things, and so they should be able to evolve independently from each other.

## 3.1 Architecture

The client can make a request to any node in the cluster. The nodes process the request and handle back the result to the client. In particular, only the primary node for a given request actually processes it. If a node receives a request for which it is not the primary, it discovers who is the primary and forwards the request to such node. The primary node processes the request and then handles back the result to the previous node, who in turn returns the response to the client.

Molto specifico, inseriscilo negli use cases?

Each node has to do three main jobs, so I created a node as a container of three services:

- *client communication service*: each node must be able to communicate with the client, to accept requests from it and return responses;
- *node communication service*: each node must be able to communicate with the other nodes. Indeed, it could have to forward the client request to the primary node, or to send the content of its own database to the backup node, or even only to check that the other nodes of the cluster are alive;
- *storage service*: each node must have a database, to store and retrieve urls on client's behalf. Actually, it has two databases: its own database, used to handle requests, and the backup database, used to replicate the data of another node. The reason for the use of two distinct databases is explained in *"The core module"* paragraph.

Figure 1 shows the architecture of the service. The communication between client and nodes exploits TCP, both because I assume an unreliable communication channel between them and because of *interactive* sessions. Indeed the possibility of starting an interactive service session between the client and a node amortizes TCP overhead, since the same amount of overhead will be payed (to open and close the connection) no matter how many times the service is invoked by the client. Instead, communication among nodes in the cluster uses UDP for the opposite reasons. First, I assume a much more reliable network inside the cluster (with respect to Internet, for instance). Second, usually single messages are sent from a node to another, and so establishing a TCP connection to send only one message would be too costly.

## 3.2 The project classes

The structure of the project is the following. The node is the building block of the cluster and, as we already said, it is made of three main *services*. Each of these services provides a high level view of the functionalities it has to implement. For the actual implementation these services rely on proper *managers*, which have the responsability to setup things and handle lower level details. Figure 5 shows a simplified UML class diagram, in which the relationships among classes are highlighted (but not the methods). In particular, this diagram only shows the classes of the *core* module, because it is this module that implements the logic of the service. Instead the *client* module only parses the user's request and sends it to proper classes of the core module to be processed.

### 3.2.1 The core module

The core module contains all the classes that implement the logic of the project. The Node class is the main class and the building block of the cluster. This module contains classes and methods to receive and process client requests, find the primary node for any request, handle the partition

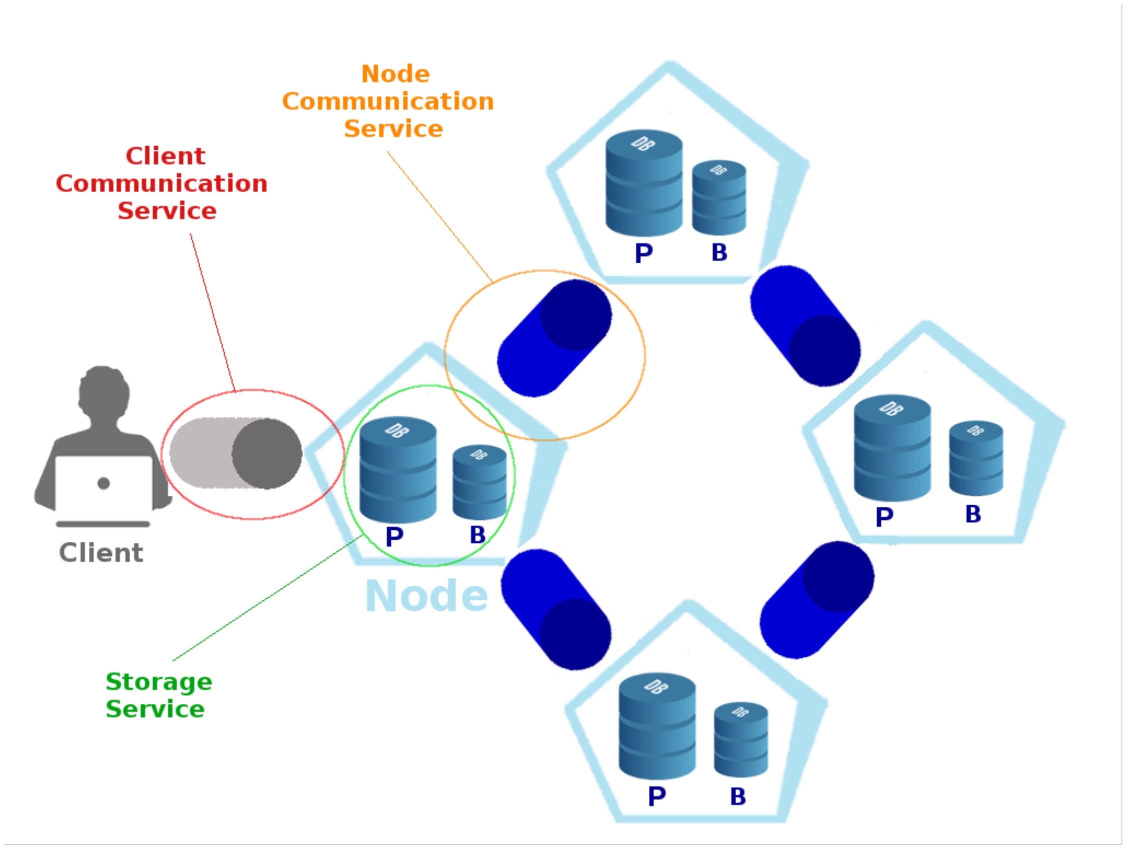


**Figure 1:** General architecture. There is a client and a cluster, which is made of more nodes. Each node is made of a node communication service, a client communication service and a storage service. The letter "P" under the database stays for "primary" , the letter "B" for "backup".

and replication of data, cope with node failure and resolve conflicts. Classes inside this module are organized into the following packages: *communication, message, storage* and *utils.* Some classes belong to the *core* package but not to any of the mentioned subpackages. I start describing these classes, and then I pass to the classes in the subpackages.

- *NodeRunner* It is the class that actually runs the url-shortener service. Indeed it setups the cluster by setting the configurations specified in the configuration file. The *main* method of the core package is in this class.
- *CoreCommandLineManager* Parses the command-line input and provides methods to deal with the options that can be specified from command-line.
- *Node* It is the building block of the whole cluster. A cluster is a collection of interacting nodes. In turn, each node is made of three services: a *ClientCommunicationService* to handle the communication with the client, a *NodeCommunicationService* to handle the communication with other nodes and a *StorageService* to deal with database operations internal to each node.
- *Service* A simple *Interface* for the various services, that only states that each service must provide a *start()* and a *shutdown()* method.

**communication** The Communication package gathers classes that are involved in the communication either with clients or with other nodes. For this reason, the classes inside this package are organized in two further subpackages *client* and *node*.

### client

- *ClientCommunicationService* One of the three fundamental services a node is made of. It starts and shuts down a ClientCommunicationManager.
- *ClientCommunicationManager* It listens on a TCP ServerSocket to accept client requests. As soon as a request is received, a ClientCommunicationThread is created and run to handle it. Then, the ClientCommunicationManager continues to listen on the socket for further requests. It also provides a method to process the client request, by delegating the processing to the NodeCommunicationManager.
- *ClientCommunicationThread* It communicates with the client by using the socket received from ClientCommunicationManager. In particular, it receives the client message, asks the ClientCommunicationManager to process it and finally returns the result to the client.

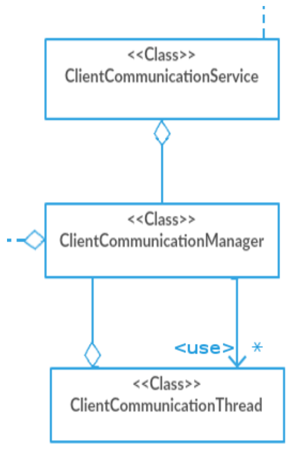

<<Class>>
ClientCommunicationService
<<Class>>
ClientCommunicationManager
<use>
*
<<Class>>
ClientCommunicationThread

**Figure 2:** UML class diagram, showing up relationships among classes in the core.communication.client subpackage

### node

- *NodeCommunicationService* One of the three fundamental services a node is made of. It starts and shuts down a NodeCommunicationManager.
- *NodeCommunicationManager* It is the responsible for the communication among nodes. It starts and shuts down the RequestManager and the ReplicaManager, and provides the capability to process the client messages it receives. In particular, if it is the primary for that request it will ask the StorageService of the node to perform the proper operations; otherwise it will discover the primary node and forward the message to it.
- *RequestManager* NodeCommunicationManager relies on this class to actually receive and send requests among nodes. A Request is intended as any message that can be received or sent by a node. It can be a client message (the message sent by the client as it is), a node message

(a message generated by the nodes that extends the client message with features like vector clocks) or an update message (a message periodically sent from a node to is backup node to update the backup database). The RequestManager acts both as server and client for the request: to manage incoming requests it spawns a RequestServerThread while to make outgoing requests it provides a *sendMessage()* method that spawns a RequestClientThread to do the job. It continously listen on a UDP socket.

- *RequestServerThread* It reconstructs the message from a stream of bytes, delegates the MessageHandler to properly process the message, collects the reply and sends it back to the requesting node via UDP.
- *RequestClientThread* This class is symmetric with respect to RequestServerThread, in the sense that it is the class that starts the message exchange with another node in the cluster.
- *ReplicaManager* Periodically sends the content of the database to the backup node. If we look at nodes in the cluster as points in a ring, the node to whom send the database is the next node clockwise in the ring. To update the backup database the ReplicaManager splits its own database in fixed-size pieces: each of these piece will be encapsulated in an UpdateMessage and sent to the backup node. To send these update messages the ReplicaManager relies on the RequestManager, because an update message is simply seen as a request.

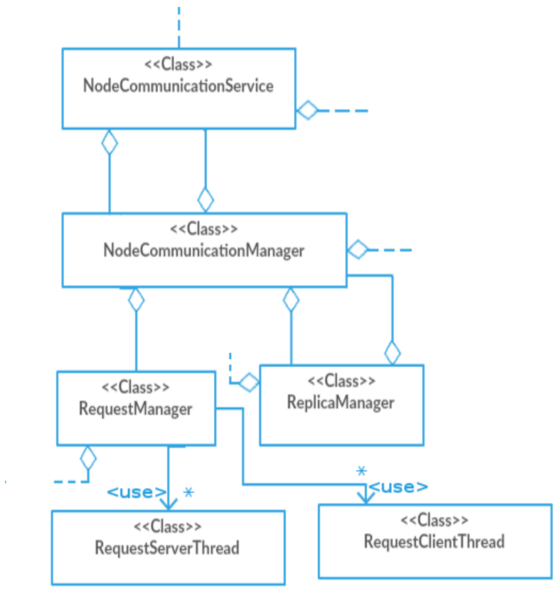
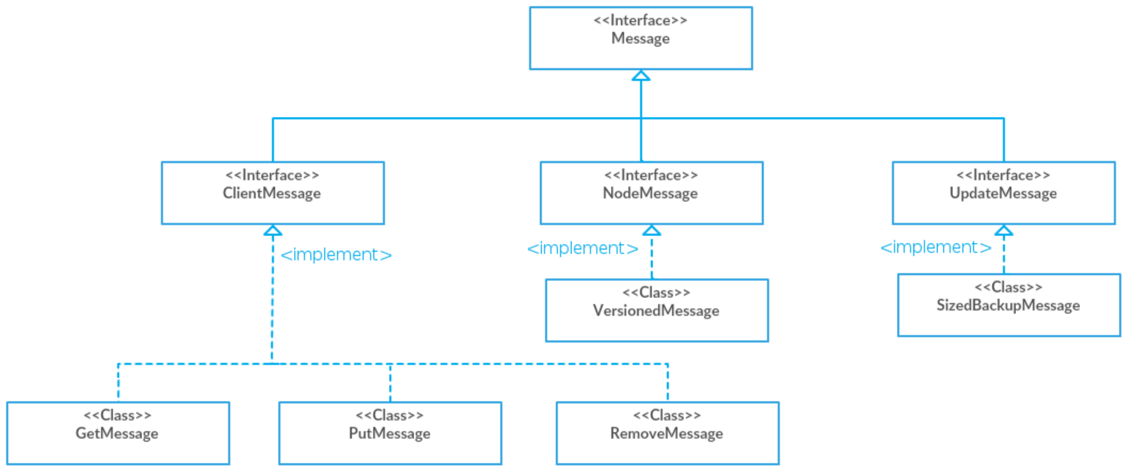


**Figure 3:** UML class diagram, showing up relationships among classes in the core.communication.node subpackage

**message** A Message is the base interface for communications. Following the layout of the project, three further interfaces are defined, that inherits from Message: ClientMessage, NodeMessage and UpdateMessage, to deal respectively with client messages, node messages and database backup.

- *Message* Base interface: inplementing classes must provide methods to return the status (success, error, ...) and the type (put, get, ...) of the message.

- *ClientMessage* Interface for client message: implementing classes must provide a method to get the url inserted by the client.
- *GetMessage* Class that implements ClientMessage, it is used when GET requests are invoked by the client.
- *PutMessage* Class that implements ClientMessage, it is used when PUT requests are invoked by the client.
- *RemoveMessage* Class that implements ClientMessage, it is used when REMOVE requests are invoked by the client.
- *NodeMessage* Interface for the messages generated by the nodes from the client messages. Implementing classes must provide methods to retrieve the original url, the associated shortened url and the vector clock associated to those urls.
- *VersionedMessage* Class that implements the NodeMessage interface.
- *UpdateMessage* Interface used when portions of the database are sent to the backup node. Implementing classes must provide methods to put and get entries of the database in the message, check whether the UpdateMessage is full or empty, return the identifier of the sender node and tell whether this message is the first of a sequence of UpdateMessage. Indeed, the content of a backup is delivered by sending a sequence of UpdateMessages, and the first message may need some special treatment.
- *SizedBackupMessage* Class that implements the UpdateMessage interface. Its name means that it is involved in backup operations and that can carry only a limited amount of database entries (indeed these messages have to be sent over UDP).



<<Interface>> Message
<<Interface>> ClientMessage
<<Interface>> NodeMessage
<<Interface>> UpdateMessage
<implement>
<implement>
<implement>
<<Class>> VersionedMessage
<<Class>> SizedBackupMessage
<<Class>> GetMessage
<<Class>> PutMessage
<<Class>> RemoveMessage

**Figure 4:** UML class diagram, showing up relationships among classes in the core.message subpackage

**storage** The classes inside this package deals with database operations.

- *StorageService* It starts and shuts down the StorageManager. It is not directly involved in any operation concerning the database.

- *StorageManager* It is the class that actually manages the database. The database is a simple *(key,value)* store, whose keys are *short urls* and whose values are *versioned long urls*, that is the original url paired with a vector clock. Each node has two databases: a primary and a backup database. The primary database is the one that contains the urls for which the node is the primary. So every time the StorageManager is asked to answer to a client request, it looks in the primary database. The backup database instead contains a copy of the primary database of another node. Namely, according to the chosen replication strategy, the backup database of node *i* contains the copy of the primary database of node *(i-1) mod N*, where N is the cluster size. The backup database is never used to answer clients' query. The backup database of node *i* is periodically updated by node *(i-1)* through the sending of a sequence of UpdateMessages. Upon receiving this messages, the recipient node checks whether the sending node is the same of the last time (this is possible because the UpdateMessage has a field that stores the id of the sending node). If it the same, the backup database is updated and everything is fine; if it is different, it means that either node *(i-1)* crashed or recovered from crashing (I assume that new nodes cannot join the cluster, but obviously crashed or partitioned nodes can re-join it). In this case node *i* merges the backup database into the primary database, because in case of crashing or partitioning of node *(i-1)*, it becomes the primary also for the urls of the crashed node. So the system is partition tolerant, because it keeps working despite the failure of a node in the cluster. When node *(i-1)* re-joins the cluster, the situation previous to the failure is re-established, with node *i* storing again the backup database of node *(i-1) mod N*. A StorageManager provides methods to read and write single entries of the database, make a dump of the primary database and merge the backup database into the primary.

**utils** This package contains classes that provide methods that don't fall in one of the previous categories, but may be needed by different classes.

- *Utils* Provides static methods to serialize and deserialize an Object and to shorten the original url. The url-shortener method exploits the 32 bit MurmurHash3 function.
- *MessageHandler* Provides static methods to process the messages. Those methods always require the StorageService to be passed as parameter, because some operation will be done on the database. Which operation depends on the type of message: PUT, GET, REMOVE and UPDATE messages get a different treatment. If MessageHandler receives a PUT message, it stores a pair *(key,value)* in the primary database; if it receives a GET message a pair is read from the primary database; if it receives a REMOVE message a pair is removed from the primary database; if it receives an UPDATE message the backup database is updated with the entries contained in the message.
- *Partitioner* The Partitioner uses the ConsistentHasher class to map the urls into a space which is partitioned among nodes, such that each node will manage the urls that follow in its partition. This is what is called ConsistentHashing. The ConsistentHasher class allows us to define a given number of virtual instances per node. This means that if we decide to give 50 virtual instances to a node, that node will be assigned 50 different (and smaller) partitions. Using this approach is good to equally distribute urls among nodes. Another strength point of the use of virtual nodes is that if a node is more powerful than the others we can assign more virtual instances to it, and this will result in an easy but effective kind of load balancing.

### 3.2.2 The client module

This module contains the classes that implement the client. The client chooses a node of the cluster and invokes an operation among PUT, GET and REMOVE. The client can establish a connection with a node that only lasts the time needed to carry out the requested operation, or an interactive session can be started, during which the connection is kept alive and the client can make as many requests as it wishes. In the latter case is the client that decides when to close the communication. The exchange of messages between the client and the cluster is done on top of TCP. The classes belonging to this module are the following:

**Client** A simple interface for the client, stating that implementing classes must provide the method *sendRequest(Message msg)* to send the request to a node of the cluster. The policy used to select the node is implementation-dependent.

**RandomClient** A class that implements the ClientInterface. Every time RandomClient has to send a message to a node, the recipient node is randomly selected among the nodes of the cluster. It uses a ClientConfig to get the list of nodes comprising the cluster.

**ClientConfig** This class holds configuration parameters for the client. In particular, it parses the configuration file to retrieve the addresses of the nodes comprising the cluster. Such addresses are stored in a list, which is returned to the client upon request.

**CommandLineManager** It parses the command line arguments, checking that nothing is missing and that the provided options are legal. Provides methods that ClientRunner uses to check the presence of such options and to display help or error messages.

**ClientRunner** Runs the client, either in batch or interactive mode, and returns the result.

## 3.3 Use cases

In this section I show how the objects of the project cooperate in some common use case. Actually there is one general use case, that is the invocation of an API by the client. Indeed, despite all the possible configurations (interactive or batch, use default or custom configuration file, ...), invoking an operation among PUT, GET and REMOVE is the only thing a user can do. All the other things that the system does, like keeping updated the backup database, being partition-tolerant, resolve conflicts and so on, are transparent to the user and so do not fall in the use cases. Instead, they will fall in the *Test* section, because we have to guarantee that those features of the system work as expected.

### 3.3.1 Processing of a PUT request

We analyze how the objects interact in case of a PUT request from a Client. The choice of PUT is arbitrary, since the pattern of message exchanges and method invocations is almost the same also with GET or REMOVE requests. The only thing that changes is obviously the operation on the database (otherwise we would have three identical operations!). Let's refer to figure 6. Although we are dealing with use cases, I will not use a use case diagram because it is too high level. Instead, I want to show how the objects interact to process a client request, and so I will take advantage of a sequence diagram. We can see that three actors are involved in the requst processing: the client, a first node and a second node. Indeed, we consider the most general case in which the client contacts a node which is not the primary for the given url. So let's start reading the diagram. The ClientRunner parses the command-line to get the request together with the original url, wraps them in a ClientMessage and invokes the *sendRquest(cmsg)* method on the Client, which result in opening a socket to send the message to the ClientCommunicationManager of Node 1. The ClientCommunicationManager, which is waiting for connections on a Socket, instantiates and

run a ClientCommunicationThread to handle the request, passing it a reference to itself. This reference is exploited for the callback of the method *processMessage(cmsg)*, which in turn results in the invocation of method *processClientMessage(cmsg)* of NodeCommunicationManager. NodeCommunicationManager creates the short url from the original url carried in the ClientMessage, then uses this short url to discover who is the primary node. At this point it discovers that it is not the primary node: so it creates a NodeMessage carrying the original url, the short url and a vector clock. Finally it instantiates a RequestClientThread to send this NodeMessage to the primary node. This message is received by the RequestManager of Node 2 (that is the primary node), which is always listening for NodeMessages (or UpdateMessages) on a socket. The RequestManager instantiates and run a RequestServerThread to process the NodeMessage, which invokes the method *handleMessage(nmsg)* on MessageHandler. This method discovers that it is dealing with a PUT request, and so it invokes the private method *processPutMessage(nmsg)*, which exploits the StorageService to store the pair *(short url, original url)* in the primary database. If the store is successful, a message with type=REPLY and status=SUCCESSFUL will be returned to the client, following back the chain of invocations.
The figure 6 is a bit simplified, for instance the signatures of the methods are not complete, the StorageService does not directly provides a *store* method (which is offered by StoreManager), and new threads are not directly created (instead it is used an ExecutorService to improve performances). I thought that these further details would have made more complex the diagram without carrying more benefits, so I have preferred a simpler but clearer diagram.

#### 3.3.2 Replication of data

## 4 Test

#### 4.0.3 Correct behaviour of put, get, remove

#### 4.0.4 Self-healing

(Primary failure) Show that in case of primary failure the system still works correctly, without needing any manual intervention

#### 4.0.5 Conflict Resolution

#### 4.0.6 Load Balancing

Tweak the number of virtual nodes to show that more powerful machines can actually be assigned more items

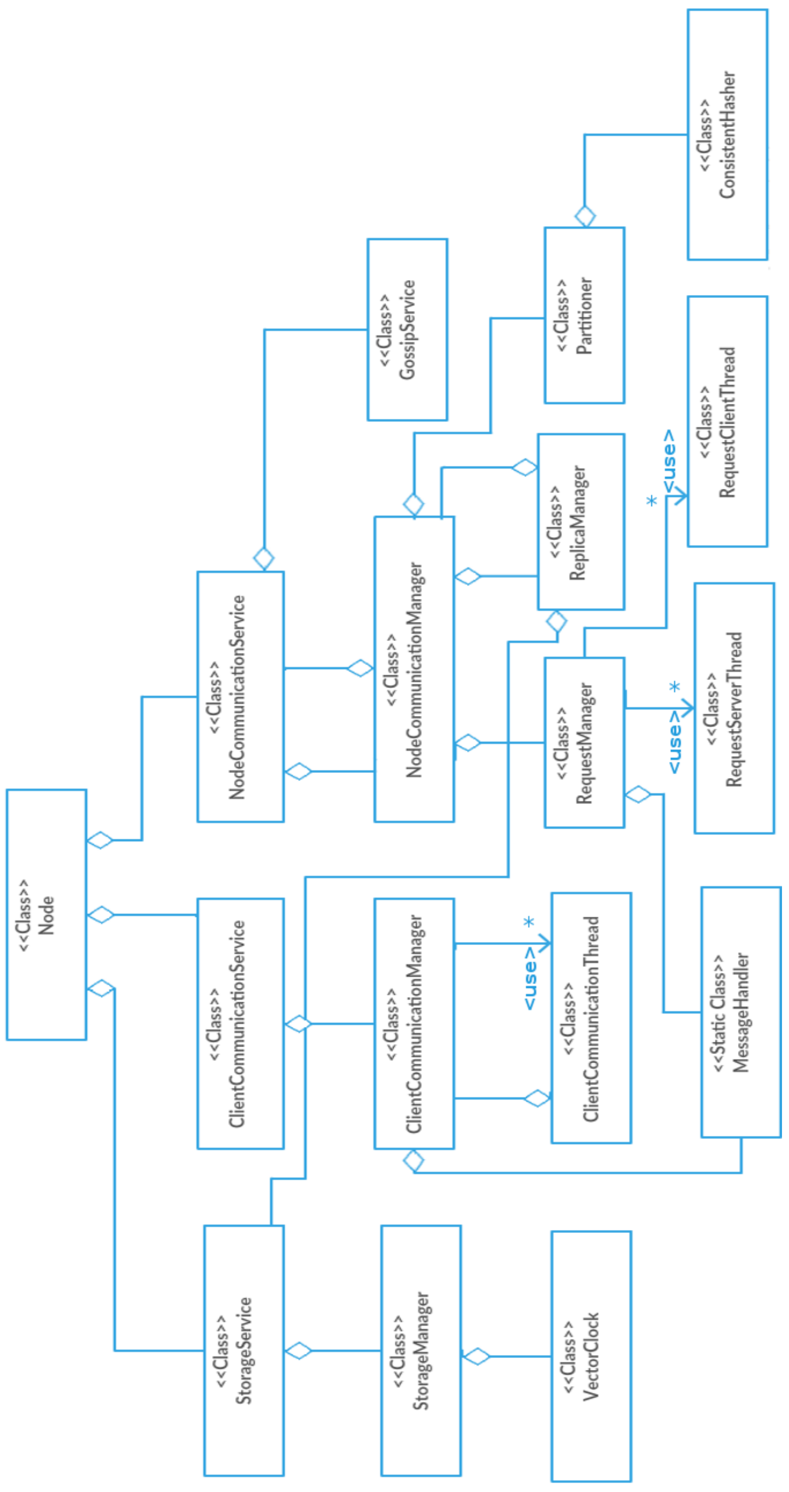
## 5 Future work

- Hash function for the url shortener.

  32 bit hash function allows too few urls; and in that case find another method to translate sequence of bytes into allowed charactes [A-Za-z0-9]

- Replication Strategy.

Every time the backup database has to be updated, the whole DB is emptied and then rebuild from scratch. This is a simple approach (the sending node sends all its database and not only the entries that have changed from the last DB delivery) but not very efficient.



<<Class>> Node
<<Class>> NodeCommunicationService
<<Class>> ClientCommunicationService
<<Class>> StorageService
<<Class>> GossipService
<<Class>> NodeCommunicationManager
<<Class>> ClientCommunicationManager
<<Class>> StorageManager
<<Class>> Partitioner
<<Class>> ReplicaManager
<<Class>> RequestManager
<<Class>> ClientCommunicationThread
<<Class>> VectorClock
<<Class>> ConsistentHasher
<<Class>> RequestClientThread
<<Class>> RequestServerThread
<<Static Class>> MessageHandler
<use>
*
<use>
*
<use>
*

**Figure 5:** Uml Class Diagram of the core module. It is shown the internal composition of the node, which is the building block of the cluster. A node is made of a NodeCommunicationService, a ClientCommunicationService and a StorageService. In turn, these services rely on proper managers, who actually implement the functionalities of the services. Where omitted, cardinalities are 1.

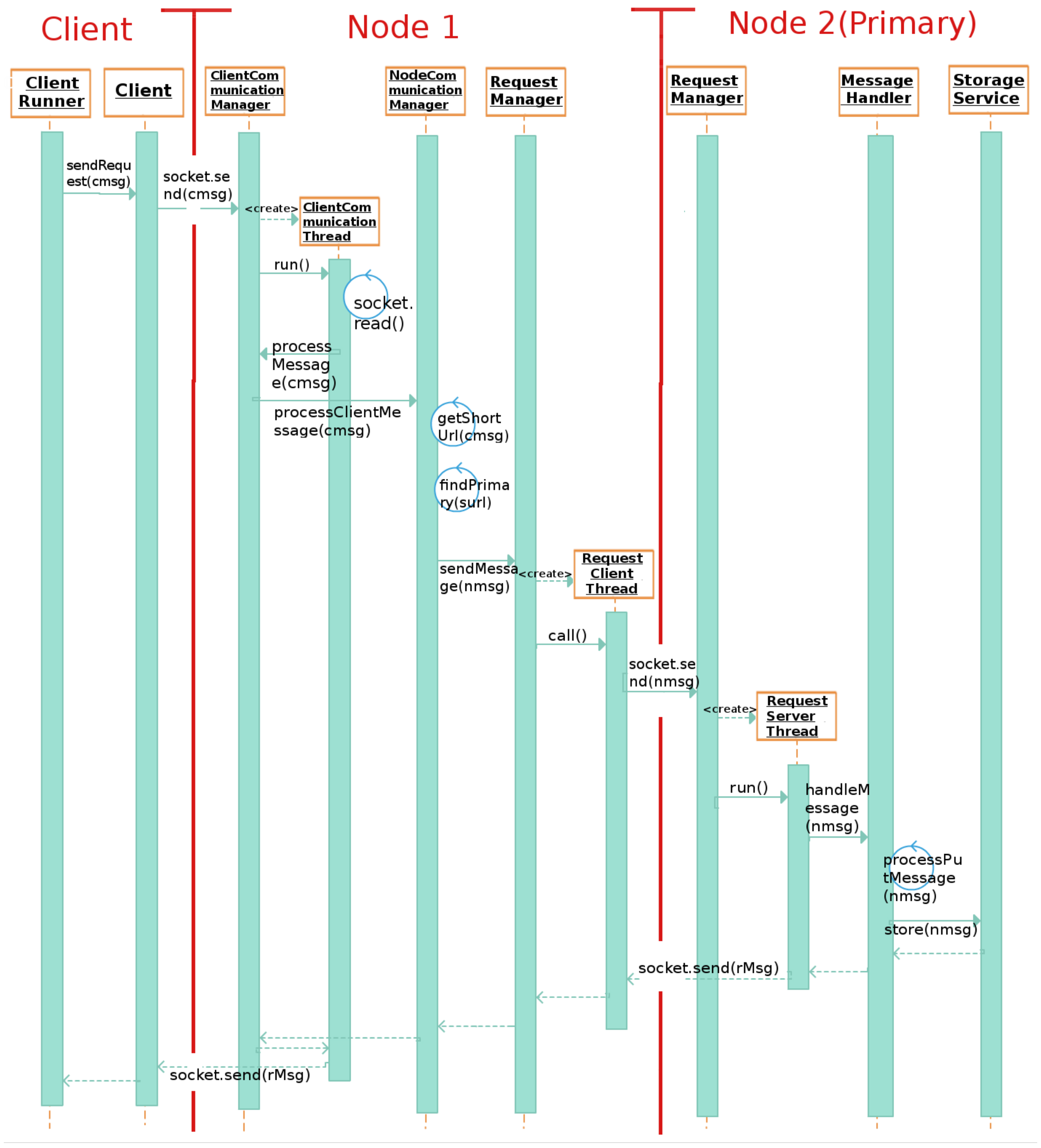


**Figure 6:** Sequence diagram for the processing of a PUT request, when the node contacted by the client is not the primary.