URL-Shortener

Francesco Balzano

April 23, 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.
Each node has to do three main jobs, so I created a node as a container of three services:

Molto specifico, inseriscilo negli use cases?

- *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.

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 2 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 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.

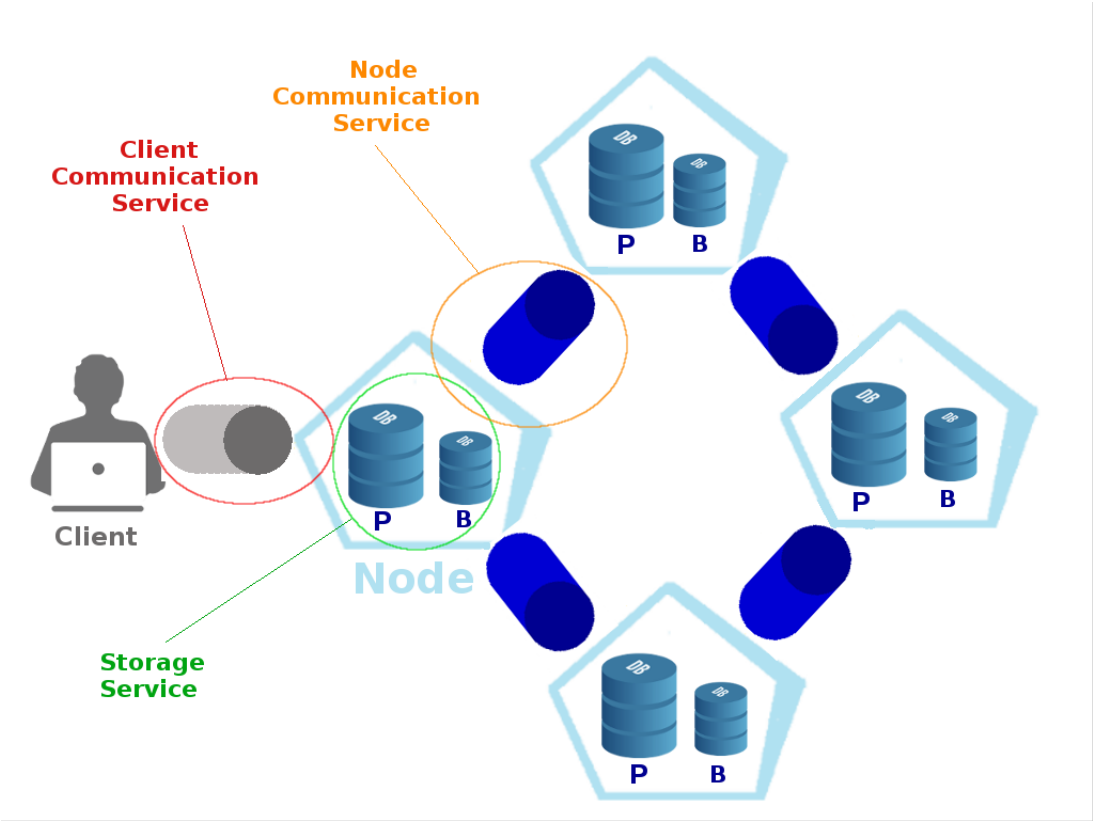

Figure 1: General architecture. There is a client and a cluster, which is made of more nodes. Each node is made of a node communication service, a client communication service and a storage service. The letter "P" under the database stays for "primary" , the letter "B" for "backup".

- *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 communication either with clients or with other nodes. For this reason, the classes inside this package are organized in two further subpackages *client* and *core*.

Client

3.2.2 The client module

3.3 Use cases

3.3.1 Management of a client request

3.3.2 Replication of data

4 Test

4.0.1 Correct behaviour of put, get, remove

4.0.2 Self-healing

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

4.0.3 Conflict Resolution

4.0.4 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]

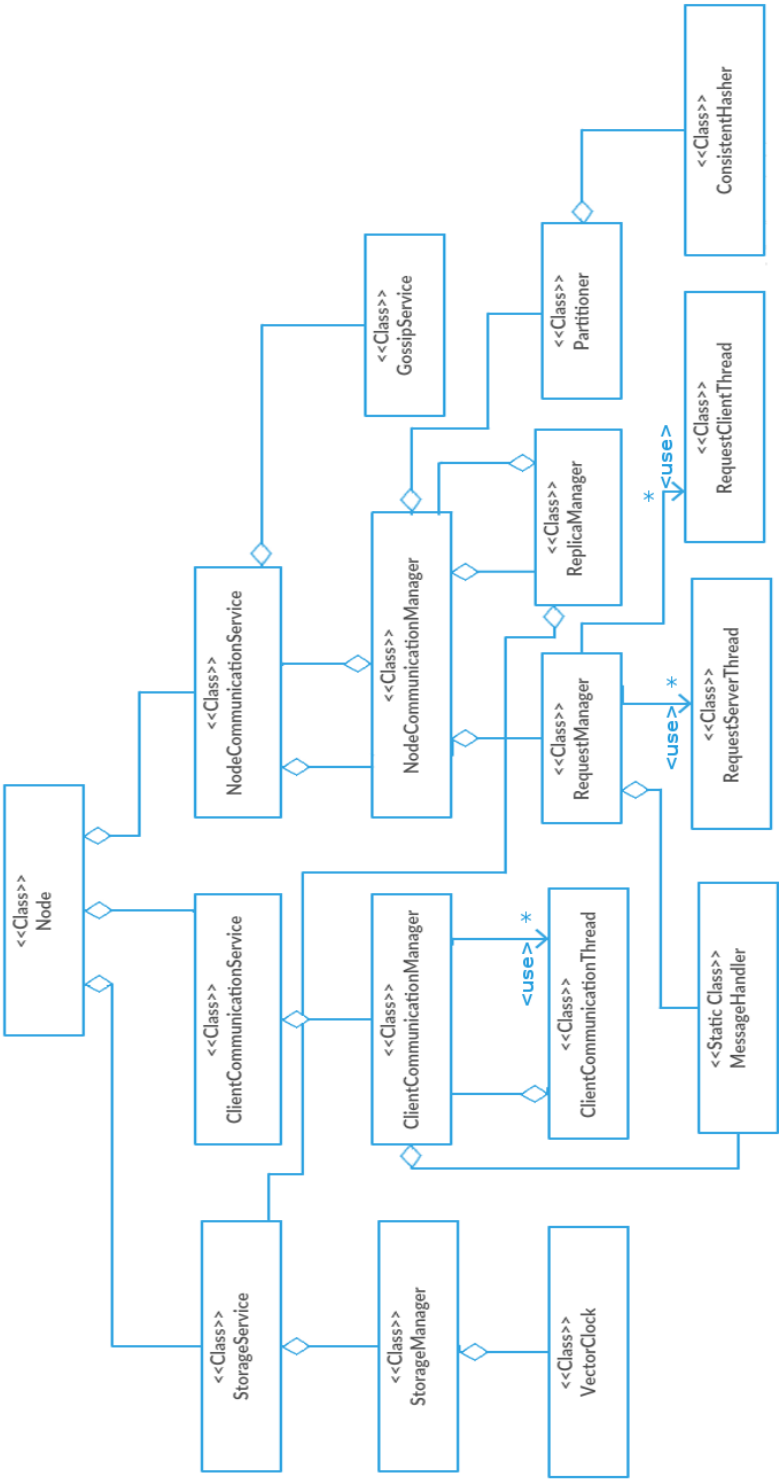

Figure 2: Uml Class Diagram of the core module. It is shown the internal composition of the node, which is the building block of the cluster. A node is made of a NodeCommunicationService, a ClientCommunicationService and a StorageService. In turn, these services rely on proper managers, who actually implement the functionalities of the services. Where omitted, cardinalities are 1.