URL-Shortener

Francesco Balzano

March 19, 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. In case of WRITE operations, the node forwards the request to the primary, which executes it and sends back the response to the sending node, which in turn hands it back to the client.
In case of READ operation, the node that receives the request firstly checks its local store: if possible it directly answers the client (running the risk of providing inconsistent information, but answering more quickly), otherwise it forwards the READ request to the primary.

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. 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 2 backup nodes, which are the next nodes clockwise in the ring. Every time a primary wishes to update its replicas, it sends data to these 2 nodes.

2.5 Primary Failure

The use of Consistent Hashing and the the specific data replication strategy adopted (backups are the next nodes clockwise in the ring) allows to avoid the need for a specific failover procedure. The failover procedure is instead automatic: 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.

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.

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. In this case, we rely on client to resolve the inconsistency, because it should know the semantics of the data it is using.