Distributed Enabling Platforms project

Ludwik Bukowski

February 22, 2018

Abstract

In this paper I'd like to introduce my evaluation of my semester project from module Distributed Enablind Platforms. The idea was to create a URL shortener based on distributed database with multinode archutecture. Firstly, I'd like to introduce the technology I used for the project. Then I will show my base approach and why it wasn't good. In next paragraphs I'd like to describe final structure and implementation notes. Finally, I will present the result and conclusions.

1 Technology

The project is implemented fully in Java language. For dependency management and building I combined *Maven*[1] and *Gradle*[2] tools. They give nice modularity and flexibility I needed. In order to separate the back-end logic and fronted I decided to use *Spring Framework*[3]. It's huge platform with gives support for many external services integration. Fron-tend side was implemented using *Thymeleaf*[4]. I decided to use this component since I have used it in past and it did very well. The tricky part was to choose the weapon for inter-node communication. I didn't want to reinvent the wheel, I used ready compontent for that case which is *RabbitMQ*[5] queues. The instance of it I run in docker container[6] and make sure every java node connects to it.

2 The problem

2.1 The use case

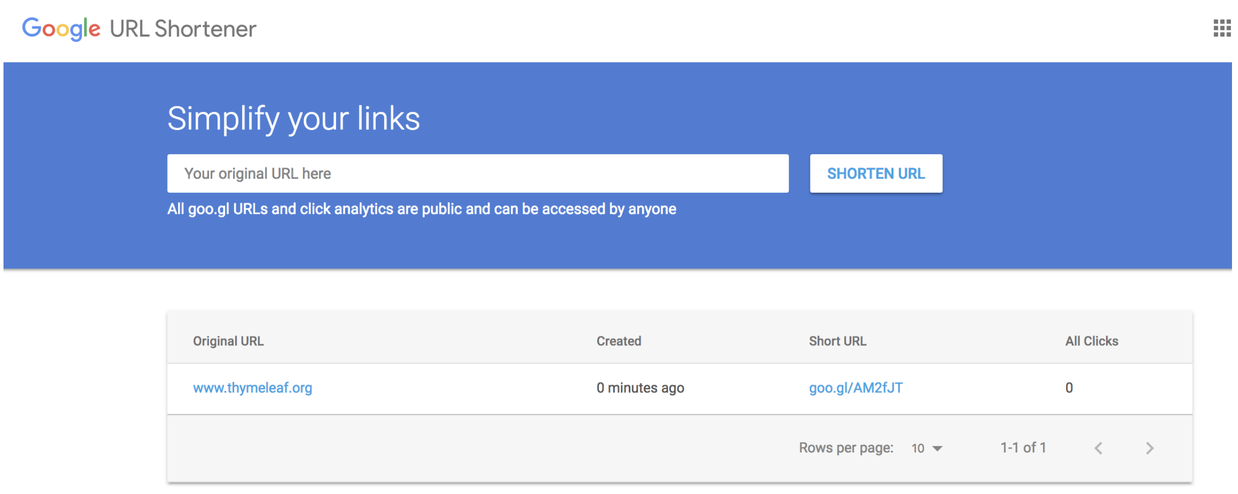
Firstly, I needed to think how my URL shortener would work. Let's forget about multi-node architecture for a second. The main idea is that the user types the *URL* and expects that the result will be shorter than the input and will always redirect to the expected page. In order to achieve that, for each *URL* we need to create a new REST resource in our platform. So, let assume that user wants to shorten the `https://www.unipi.it/index.php/studenti`. The output result will be `http://us?x=4E05D0`.The point is that we generate brand-new *URL* located on our *"us"* domain. When we click on this link, it redirect us to students section on University of Pisa page. How do I resole it? For input *URL* we generate short hash string. Then, the system just keeps key-value pairs for each record. The key is the generated hash for *URL* and the value is the root page. When we'd like to find the page for specific short *URL*, we just do simple lookup in our internal DB. Simple Google's example can be seen on *figure 1*. The main database operations supported are due to simplified version of the *CRUD* rules which is:

1. Put
2. Remove
3. Read

However, we are not satisfied with just the local storage, we'd like to go for distribution.

2.2 The Communication

The main idea behind this project that we want our platform to be scalable and reliable, in other words - distributed. Firstly, we need to specify how we want our nodes to communicate. As


Google URL Shortener

Simplify your links

Your original URL here

SHORTEN URL

All goo.gl URLs and click analytics are public and can be accessed by anyone

Original URL	Created	Short URL	All Clicks
www.thymeleaf.org	0 minutes ago	goo.gl/AM2fJT	0

Rows per page: 10 1-1 of 1

Figure 1: Simple example of generating new short *URL* for specific web page

previously written, I use *RabbitMQ* queues. Every instance keep the connection with other nodes' queues. If we want to send a message to other node, we just simply serialize it and put in specific queue. Then, the receiver consumes the message asynchronously. The basic idea shown on the *figure 2*. Keeping in mind that *RabbitMQ* provides us reliable tool for message exchange, we can go on to the distributed database part.

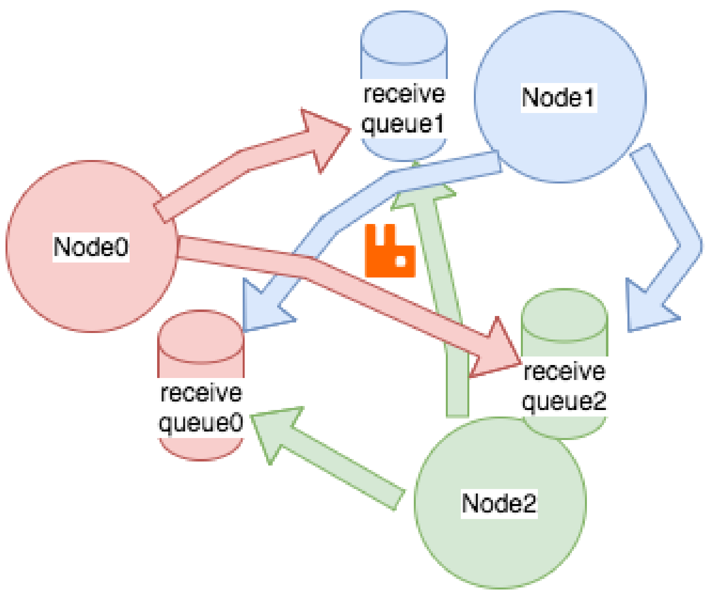


Figure 2: Three nodes exchanging the vector clock list

2.3 The Synchronization

In order to keep the proper ordering of the events on each node, I decided to use *Vector clocks algorithm*[7]. The concept is quite simple - we keep a list of counters(each counter per each node). When the event occurs, we increment local counter and we distribute the whole vector clock list to other machines. The task to be done by others, is to merge local vector clock with the incoming one and update the local one.

The simple concept shown on the *figure 3*. Basically, we want to synchronize our nodes, so keep the same ordering of the events everywhere.

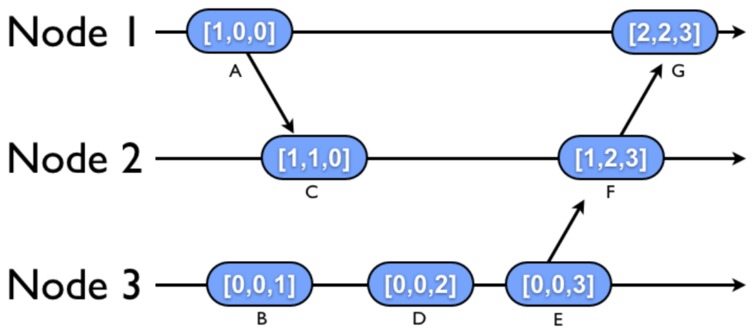


Figure 3: The queues as a base communication platform. On the figure we can see 3 nodes, communicating with each other

2.4 The Distribution - legacy idea

For now, we know how works the local storage, we are sure about our inter-node communication and even more - we are able to synchronize them correctly. The next step was to introduce the storage architecture.
My first idea was to keep whole data replica on each node. Then, every *Put* and *Remove* requests were distributed across the system. The *Read's* were "dirty", which means that we don't compare the value with rest of the system and do just local read. Moreover, every instance kept not only database in the RAM but also the message history was stored on the local disk so that it could be iteratively restored when the node died. The advantage of such a solution might be that the system still works when there is at least one node running and the reads are fast. Nevertheless, there are more disadvantages. Firstly, the storage is limited and it cannot be scaled horizontally. Other thing is the fact, that the message history does not give us any value for that particular URL shortener use case. The last bad issue are the reads - they are not reliable and such a implementation might end up with many race conditions.

2.5 The Distribution - new approach

Hence, I needed to introduce another, completely different approach. I have chosen (after my Professor's advice) the *Consistent hashing technique*[8]. We don't store whole database on each node anymore - we distribute the values, based on the keys' code hash. The idea is actually quite simple - we divide the keys' domain between machines. Each instance gets it's own range.(see *figure 4*). Let's say we have machines 0,1 and 2. When new request to machine 0 occurs, we compute the

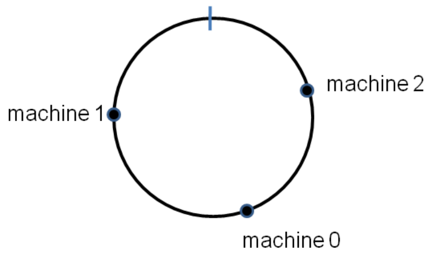


Figure 4: We divide the possible key's domain among the three machines

code for the key. If it fits machine0's range its fine and we just store the value locally and inform (through *vector clocks*) other nodes about the event. The more interesting is the case, when the computed code is supposed to be managed by machine1. Then, we have to forward the *Put* request there so that it can be stored. The concept is shown on *figure* 5. Exactly the same approach is taken for the *Remove* request. A bit different steps are done for the *Read*. If the wanted key is stored on the requested machine, we just return the value. We don't inform other nodes about this event. However, when the hash for the key is stored on other machine (lets say machine2), we need to do the synchronous call for the value. Read procedure explained on the *figure* 6.

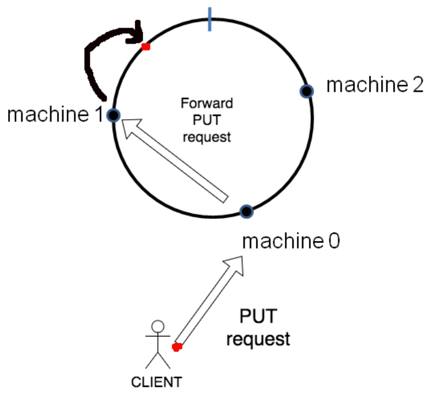


Figure 5: machine0 passes the PUT request to machine1

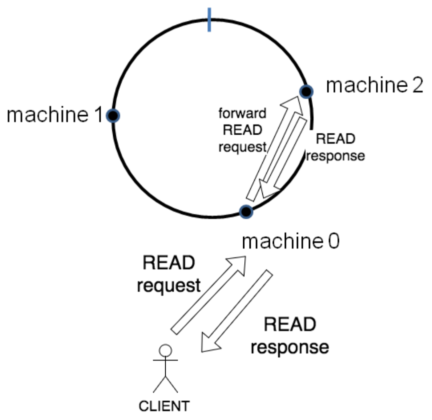


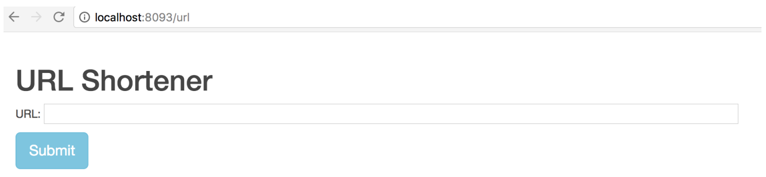
Figure 6: machine0 passes read request to machine1. Please pay attention to synchronous nature of this flow.

3 The result

Basically, I have tested my system with three java nodes. One each of them, exposes the HTTP endpoint to which user is able to connect through HTTP connection. On the *figure* 7 we can see the main web page. The user is supposed to type the *URL* to shorten in the area provided. Then, he clicks on *Submit* button and end up with the second page with new link generated (*figure* 8). At this stage, user is able to remove links as well.

4 Corner cases

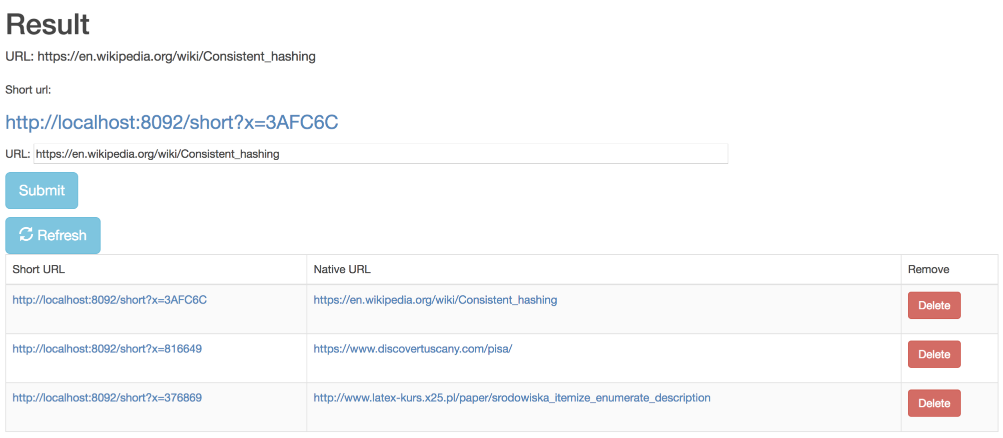
When talking about distributed systems we need to think about all possible bad situations that might happen. Firstly, when the node which contains the data we'd like to read is down, my approach is to wait some timeout and inform client that the data is not available. Secondly, when



URL Shortener

URL:

Figure 7: URL shortener main page



Result

URL: https://en.wikipedia.org/wiki/Consistent_hashing

Short url:

http://localhost:8092/short?x=3AFC6C

URL: https://en.wikipedia.org/wiki/Consistent_hashing

Short URL	Native URL	Remove
http://localhost:8092/short?x=3AFC6C	https://en.wikipedia.org/wiki/Consistent_hashing	Delete
http://localhost:8092/short?x=816649	https://www.discovertuscany.com/pisa/	Delete
http://localhost:8092/short?x=376869	http://www.latex-kurs.x25.pl/paper/srodowiska_itemize_enumerate_description	Delete

Figure 8: URL shortener page with shorten URLs

node is up and running after the fail, it needs to revoke the current vector clock. It will happen when first event occur in the system - the nodes will figure out that clocks are not compatible and will reset the clock on every node. In the final version, in case of *Put* or *Remove* vector clocks' conflict, the action is not applied but the vector clock is reset all across the system. It's still better to make user repeat the action instead of making the system inconsistent or/and not working. When it happens for the read requests, We follow the same steps (with vector clock reset) and repeat the read procedure. It's important, that user gets the value, even though it might be old - at the end, there are not so many cases where the key will be overwritten (assuming the hash function is good and we use big key domain).

5 Summary and improvements

During the implementation I have faced few technology-based issues. Generally, there were some problems with *RabbitMQ* library - there is no good built-in support for synchronous message consumption. There was, in previous versions but they deprecated it and there was a reason - it wasn't reliable. I needed to walk around it with custom *BlockingQueue* mechanism.
Secondly, the firstly proposed approach wasn't satisfying so the whole architecture needed to be reimplemented. It took some time, but in the end this is all about learning based on experience. There is still room for improvement.
The issue may be the nodes' failure. In this scenario, every node is responsible for the data of it's neighbor as well. Another optimization would be to use *virtual nodes*. Using *virtual nodes* will ensure that the new node soaks load from many other physical nodes rather than splitting load with just one other physical node. The challenging thing would be also the data reconfiguration when I'd like to add more nodes to existing system. Then, we need to divide the whole domain once again - the whole procedure might take time.

References

[1] https://maven.apache.org/

[2] https://gradle.org/

[3] https://spring.io/

[4] https://www.thymeleaf.org/

[5] https://www.rabbitmq.com/

[6] https://www.docker.com/

[7] https://en.wikipedia.org/wiki/Vector_clock

[8] https://en.wikipedia.org/wiki/Consistent_hashing