Distributed Operating Systems

Prepared for: Adam Kozakiewicz

Prepared by:

Ozan Oz

Karol Dzitkowski

David Miguel Lozano

Dawid Pachowski

10 Nov 2015

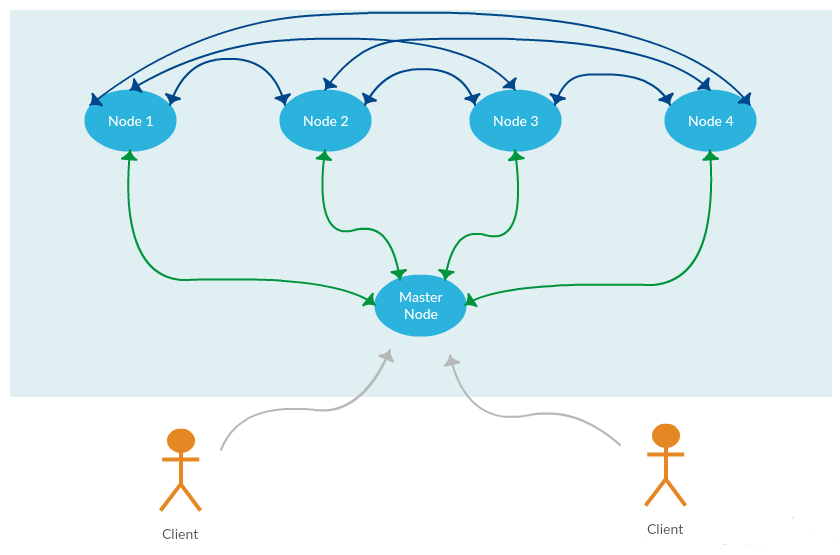
Introduction

Our goal is to create distributed database focusing on fault tolerance, ease of deployment and scalability. All the data will be replicated in several nodes, improving the performance and protecting the availability of applications. The stack of technologies that we will use will reduce the cost of deployment. Furthermore, it will enable database scale up instantly and database scale out transparently.

Our project has 2 phases. After we complete Phase 1, if there is enough time left, we will start working on Phase 2. If we do not have time for completing the second phase successfully, we will, at least, add a backup master node to the first phase.

In what follows we shall explain both phases.

Phase 1

Architecture

1- Client sends the query to the master node via command line application.

2- Master node delivers the query to a node chosen by load balancer, without parsing the query.

3- After parsing the query, slave node asks the master node to/from which nodes it should copy/read the data.

4- Master sends the answer back to the node (i.e. where is data from tables A and B, it answers A is on ip1: 12.123.123 and ip2: 13.123.123.3, B is on ip1: 191.21.123.33 or if master is asked where to put 10 000 new records to table A? Master answers: to nodes with ip 312.31.312.3, 12.123.132.31 etc.)

5- Node ask for the data to the nodes pointed out by master and collects it. Then it creates a temporary table to execute the query locally.

**Replication Factor(RF)** will be adjusted by us It will be configurable. It would be possible to set the number of copies of each piece of data up for increasing fault tolerance.

**Queries**

Our distributed operating system does not allow subqueries. Queries are handled by SQLite.

An example query ;

SELECT ID, SUM(money)

FROM A LEFT JOIN B ON A.externalId = B.Id

WHERE A.time > '12-12-2015' AND B.sth > 3;

We provide only synchronous operations for a client with monotonic reads and writes.

**Master Node**

Master node has 3 tables:

1- Master Table : Keeps information about all rows, such as; Row\_id , Table\_id , Node\_id

2- Node Table : Keeps information about all the nodes, such as ; Node\_id , Node\_ip, status

3- Query Table : Keeps information about queries, such as ; Query\_id , Node\_id , Query\_Status

Master node tables and temporary tables in slave nodes uses in memory database. Master node also gives version\_id, a precise timestamp for the new data and it is stored in slave nodes as an additional column so if we have the same row more than one, query uses the latest version.

Consistency

FIFO Consistency

All write operations from one process will be seen by all other processes in the order they were issued, but there are no constraints on the order of write operations issued by different processes.

Client Centric Consistency

**Monotonic Reads**

After a read operation returns the new value of data, no other read will return the old value. We provide this by using version numbers. In cases when there are more than one copy of the same row, we check the latest version.

**Monotonic Writes**

Writing operations are synchronous in a process, one must complete before the other writing operation starts.

**Read Your Writes**

After a successful write operation, new data is being seen by read operations. Write and read operations are synchronous by the same process.

**Writes Follow Reads**

A write operation following a read operation by the same process modifies the same or newer version of data.

For INSERT many values, node asks master for a list of nodes to insert new data to table X, master gives him a list and node asks those nodes through internal API to insert this data to their storage.

For SELECT, node asks master for the locations of the selected rows. Master answers that request. Node asks for the rows from the nodes master told him and puts them into a temporary table.

For UPDATE, node asks master for the locations of the rows he wanted to update. Master tells him which nodes have those rows, and node sends the UPDATE query to those nodes through internal API.

For DELETE, node asks master for the locations of the rows he wanted to delete. Master tells him which nodes have those rows, and node sends the DELETE query to those nodes.

After all of these operations, when this is completed the node says to master that he completed the task, master stores this information in his table and responses success to client. If something fails before, master sends failure message to client.

Major Technical Assumptions

We have a big server room with many machines

We have 1 gigabit internet between nods

We have Docker installed in our servers.

We have a lot of RAM on our servers

Technologies We Use

- Elasticsearch, Logstash, Kibana (ELK)

<https://www.elastic.co/>

- Virtual Machine

<https://www.virtualbox.org/>

- Java

<https://www.java.com/>

* SQLite

<https://www.sqlite.org/>

* Docker

<https://www.docker.com/>

- Github

<https://github.com/>

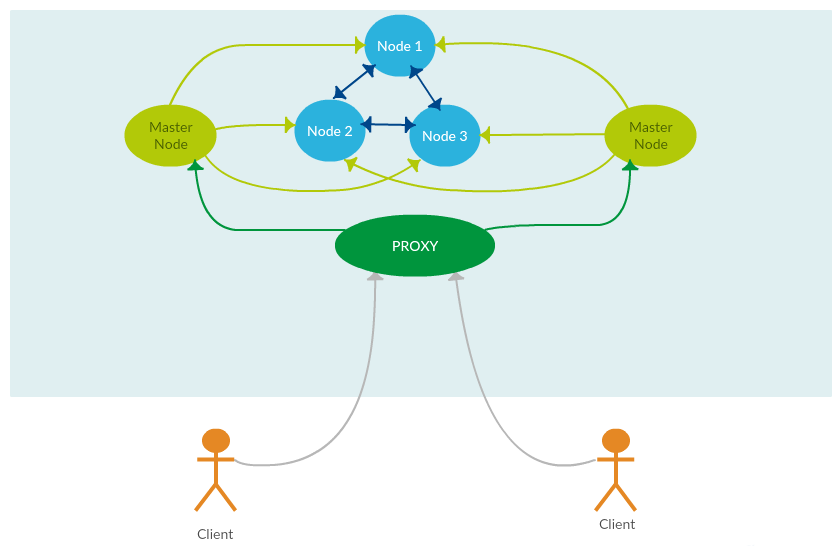
- Travis CI

<https://travis-ci.org/>

- Waffle

<https://waffle.io/>

Phase 2



The difference in this phase is we have more than one master node and a proxy. By using more than one master node, we get rid of single point of failure and increase the fault tolerance.

Our challenge in this step is setting the clock synchronization very well to avoid time stamp inconsistencies when we are updating rows in our nodes. We use very precise timestamp as some 64bit or 128bit integer and it will be a logic clock.

We are planning to use Lompart's Algorithm to set the logical clocks.

Masters will have additional tables and all masters will have to be registered in all others before starting work. Moreover they will synchronize fully their internal tables.