Faculty of Electronics and Information Technology

WARSAW UNIVERSITY OF TECHNOLOGY

Concept Solution Design
Distributed Voting System

Course: 103A-CSCSN-MSA-EDCS

Period: Summer Semester 2022/2023

Professor: Wiktor Daczuk

Student 1: Bruno Axel Kamere

Student 2: Senkoro Salome Justin

Date of Submission: 19/04/2023

ul. Nowowiejska 15/19
00-665 Warszawa
www.elka.pw.edu.pl

Table of Contents

1. Introduction

This report aims at proposing a concept solution for a distributed voting system, topic 15 of the 103A-CSCSN-MSA-EDCS summer semester 2022/2023 course. The voting system objective is; members propose a voting subject, for which they then vote on, and results announced later.

The proposed system implements various key features and characteristics of a distributed system including but not limited to fault tolerance, parallel computing, high availability, et cetera. The proposed solution will be deployed on Amazon Web Services (AWS) EC2 servers, running Docker services. The proposed solution will ensure that servers' filesystems are always kept in sync by implementing a distributed filesystem solution.

This document goes along to list out the general assumptions of the project, high-level architecture of the solution as well as a test plan to evaluate that the system is indeed distributed and has characteristics of a distributed system.

2. General Assumption

The project assumes the following:

- Voting rounds (Elections) should only be available for a defined period.
- Voters should be able to have voting options only elections are available.
- Voters should only be able to vote once.
- The election authority should set a voting topic and election period.
- The system should be:
    - Run on at least 3 servers.
    - Be scalable.
    - Be fault tolerant.
    - Be transparent. End-Users should see the system as one rather than a collection of multiple components.
    - Be heterogenous. The system should run on different types of operating systems and implement at least 2 programming languages.
- The voting application will be built with 2 programming languages:
    - Python.
    - NodeJS.
- The servers will be running 2 operating systems:
    - Ubuntu.
    - RHEL.
- The servers should be able to keep their filesystems in sync.

3. Solution Designs

3.1. High Level System Architecture & Sequence Diagram

The system is going to be built and deployed on Amazon Web Services on a cluster of 3 EC2 instances load balanced by an AWS Application Load Balancer. The EC2 instances will run Docker services in Docker swarm across the cluster where services.

On a high level, the proposed solution will consist of 5 main components:

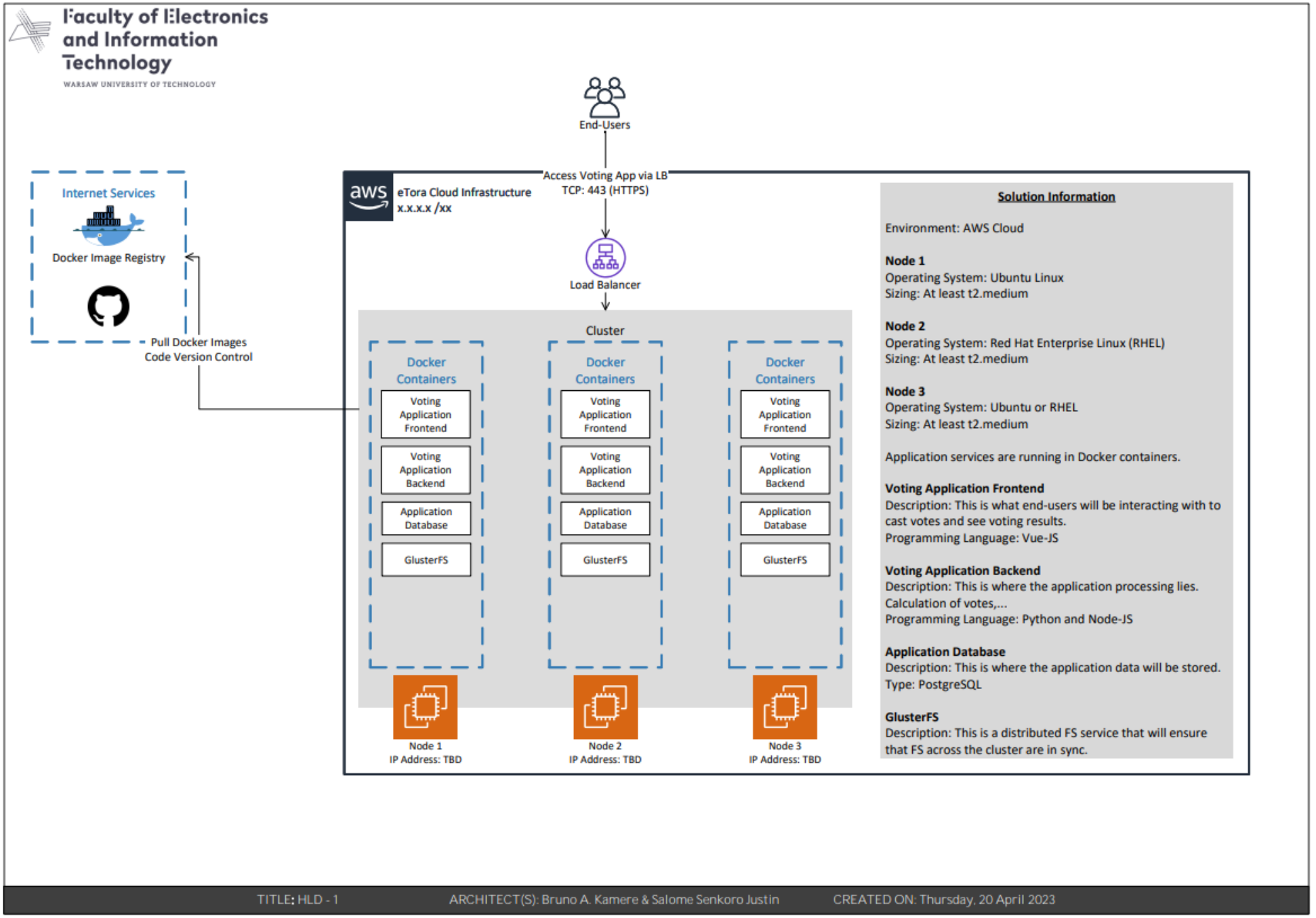
- A cluster of servers. This is where all the application services making up the solution will be deployed on.
- An Application Load Balancer. This will ensure that traffic is load balanced across the cluster in a meaningful way.
- The voting application front-end and back-end.
- An application database. This will store the application's data.
- A distributed filesystem implemented with GlusterFS. This will ensure that all nodes of the cluster have a synced filesystem.

As mentioned above, the proposed solution implements a cluster of at least 3 servers running on the AWS EC2 service. These servers need to run in a distributed manner, meaning that they need to be at least:

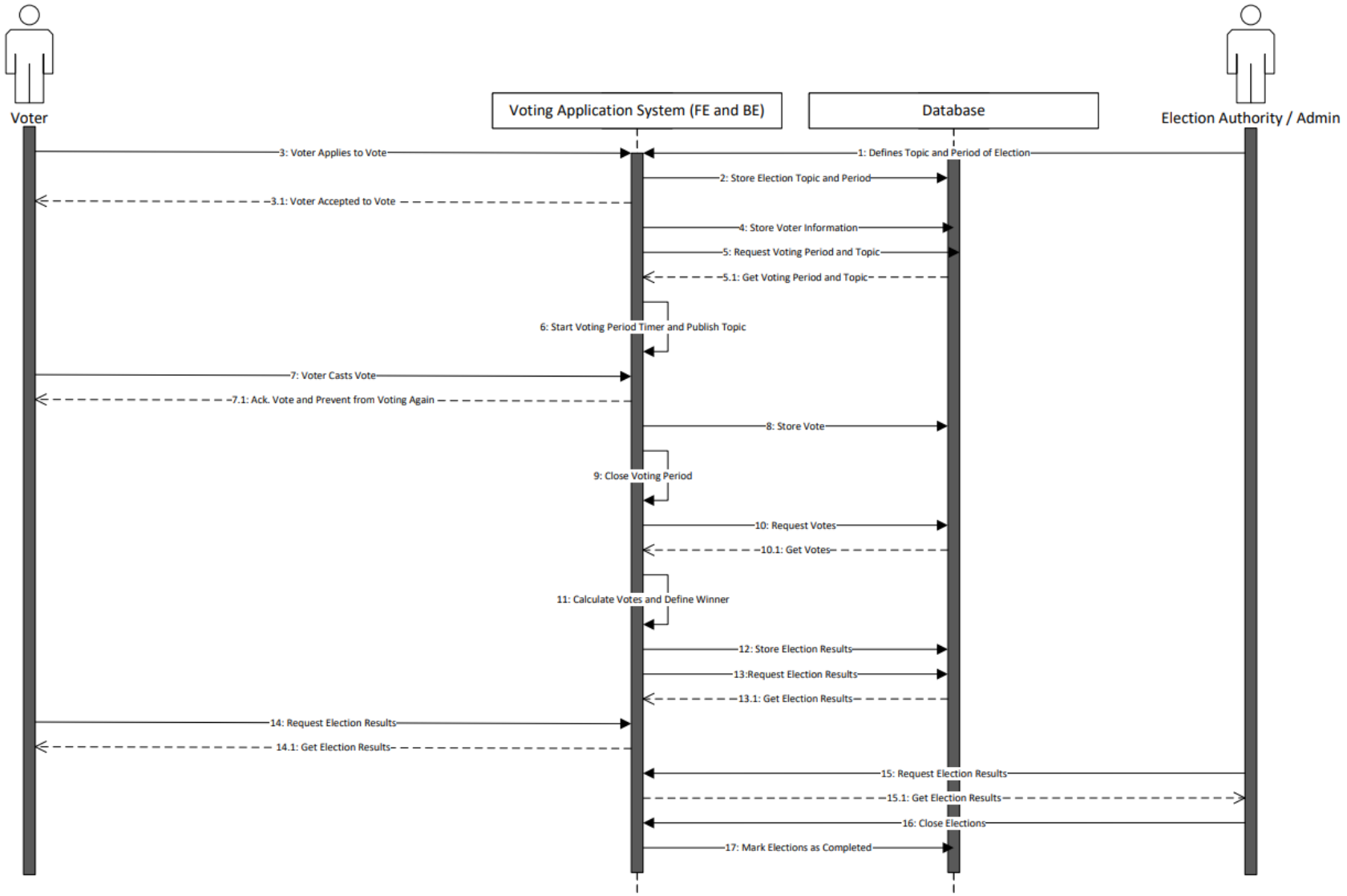
- Highly available.
- Fault tolerant.
- Scalable.
- In sync with each other.

The above characteristics need to be inherited by the Docker services deployed on the servers as well.

The images on the next pages illustrate the system architecture high level design as well as a sequence diagram.


Faculty of Electronics and Information Technology
WARSAW UNIVERSITY OF TECHNOLOGY
End-Users
Access Voting App via LB
TCP: 443 (HTTPS)
Internet Services
Docker Image Registry
Pull Docker Images
Code Version Control
aws
eTora Cloud Infrastructure
x.x.x.x /xx
Load Balancer
Cluster
Docker Containers
Voting Application Frontend
Voting Application Backend
Application Database
GlusterFS
Node 1
IP Address: TBD
Node 2
IP Address: TBD
Node 3
IP Address: TBD
Solution Information
Environment: AWS Cloud
Node 1
Operating System: Ubuntu Linux
Sizing: At least t2.medium
Node 2
Operating System: Red Hat Enterprise Linux (RHEL)
Sizing: At least t2.medium
Node 3
Operating System: Ubuntu or RHEL
Sizing: At least t2.medium
Application services are running in Docker containers.
Voting Application Frontend
Description: This is what end-users will be interacting with to cast votes and see voting results.
Programming Language: Vue-JS
Voting Application Backend
Description: This is where the application processing lies. Calculation of votes,...
Programming Language: Python and Node-JS
Application Database
Description: This is where the application data will be stored.
Type: PostgreSQL
GlusterFS
Description: This is a distributed FS service that will ensure that FS across the cluster are in sync.
TITLE: HLD - 1
ARCHITECT(S): Bruno A. Kamere & Salome Senkoro Justin
CREATED ON: Thursday, 20 April 2023

Faculty of Electronics and Information Technology
WARSAW UNIVERSITY OF TECHNOLOGY



TITLE: Sequence Diagram ARCHITECT(S): Bruno A. Kamere & Salome Senkoro Justin CREATED ON: Thursday, 20 April 2023

4. Test Plan

After the solution implementation, tests need to be conducted to ensure that the solution is indeed a distributed system. Some test cases will be run through AWS Fault Injection Simulator (AWS FIS).

AWS FIS (Fault Injection Simulator) is a service provided by AWS that enables developers to perform controlled experiments on AWS resources to test their resilience and identify weaknesses in a system. It allows developers to simulate different types of failures or disruptions that could occur in a production environment and gain insight into how applications and services would respond to these failures.

Below are test cases that will be performed:

1. Node failure.
2. Network Disconnection.
3. Proper votes calculation.
4. Elections integrity (a voter needs to only vote once during a specific period).

4.1. Node Failure

Test Steps:

1. Using AWS FIS, take down one node from the cluster.
2. Observe how the system behaves.

Expected Results:

- The system should detect this and automatically reprovision the failed node.
- Services that were running on the failed node, should still be available on the other nodes with little to no downtime.
- After a new server is reprovisioned, it needs to catchup and sync with the other nodes and reprovision necessary cluster services.

4.2. Network Disconnection

Test Steps:

1. Using AWS FIS, simulate a network disruption between nodes.
2. Update files on one server.
3. Reconnect the servers.
4. Observe how servers synchronize.

Expected Results:

- The system should be able to get back in sync and use the latest changes after connection is regained.
- Users should be able to still have access or be informed of the network disruption and shouldn't see any direct effects.

4.3. Proper Votes Calculation

Test Steps:

1. Allow multiple voters to cast their votes.
2. Observe that the application can aggregate votes from different end-users and calculate a result.

Expected Results:

- Calculations should be accurate.

4.4. Elections Integrity (a voter should only vote once)

Test Steps:

1. Let a voter vote.
2. Check if the voter still has the option to vote.

Expected Results:

- If a voter has voted, they shouldn't be able to take part in the elections again unless another voting round has started.

5. Conclusion

The proposed solution is still a concept and is subject to changes along the way. More detailed designs and documentation will be produced along the way.

Even though, the proposed solution is a concept though, it still presents a high chance of being implemented as proposed with little to no changes.