# Probabilistic CAP implementation for Latency and Consistency SLAs

Cassandra Cluster Architecture Setup

Design Specification

Version 0.1

**Submitted By:**

Dhanashree Gaonkar

dhanashree.gaonkar@sjsu.edu

7/25/2018

**Advisor**: Rakesh Ranjan

## Contents

Version History ........ 2
Introduction ........ 3
References ........ 5
Requirements ........ 6
Functional Overview ........ 7
Configuration/ External Interfaces ........ 8
Logging and Debug: ........ 8
Implementation ........ 9
Testing ........ 12
General Approach ........ 12
Unit Tests ........ 12

## Version History

| **Version** | **Changes** |
|---|---|
| 0.1 | Cassandra Cluster setup |

## Introduction

This document explains in detail about the setup and design of Cassandra cluster which is our first step towards the implementation of PCAP system. The CAP theorem was firstly introduced by Eric Brewer (2000) as a conjecture in a conference keynote. Brewer postulated that three properties (Consistency, Availability, and Partition-tolerance) are desirable from a real-world web service. A distributed system can pick at most two of these three desirable properties. Various distributed storage systems design is influenced by the CAP theorem. Abadi, D. (2010) stated in a blog that "CAP falls far short of giving a complete picture of the engineering tradeoffs behind building scalable, distributed systems". In reality, partition-tolerance is inevitable, so the system must choose between CP and AP during a network partition. Our Project implementation tries to do the same by creating a layer on top of the NoSQL database. This will allow the application pushing/ pulling data from the database to dynamically switch between CP or AP in a probabilistic manner. We have chosen Cassandra as a test database and the first part of implementation involves setting up a Cassandra cluster architecture.

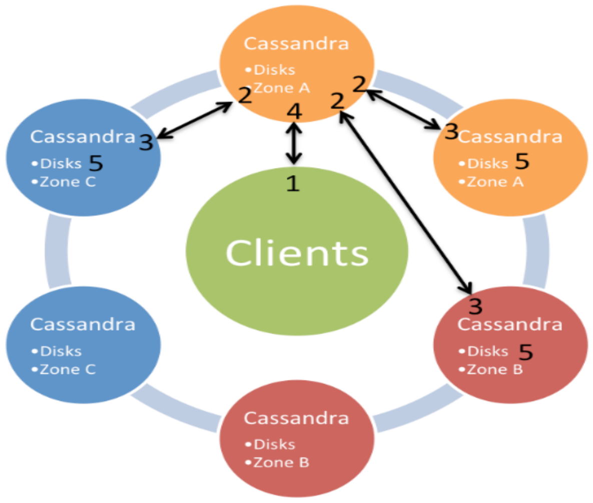


Fig 1. Example of a Cassandra Cluster. Adapted from 'https://cdn.intellipaat.com/wp-content/uploads/2015/08/img-1.png'

Cassandra is a distributed database by nature. It was designed to run in a network of computer nodes as a server with distinct parts running on different machines. The nodes' coordination and data distribution are inside its own architecture. This is one of the reasons that a Cassandra network is easier to scale horizontally than other common relational database systems.

The typical Cassandra network topology is composed of a cluster of nodes, also called a Cassandra ring, running in different network addresses located on different physical servers. In the above given figure, the client can write to any node in the cluster. There is another node called Coordinator which will replicate the data to all the other nodes and zones. Once the coordinator returns acknowledgement to the client the data is written to log in the internal commit log file.

If a node goes offline, hinted handoff completes the write when the node comes back up. Requests can wait for the acknowledgment of a single node/ quorum of nodes / all  nodes.

## References

[1] CassandraDB (https://hackernoon.com/using-apache-cassandra-a-few-things-before-you-start-ac599926e4b8)

[2] CAP theorem blog (http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter)

[3] Deployment on AWS (https://blog.bluesoftglobal.com/set-cassandra-aws-solid-ha/)

[4] NoSQL Wiki (https://en.wikipedia.org/wiki/NoSQL)

[5] CAP Theorem explanation (http://robertgreiner.com/2014/06/cap-theorem-explained/)

[6] Abadi, D. (2010). Problems with CAP, and Yahoo's little known NoSQL system [Blog Post]. Retrieved from http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-andyahoos-little.html.

[7] Brewer, E. A. (2000). Towards robust distributed systems (abstract). Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00. doi:10.1145/343477.343502.

[8] Cassandra Architecture (https://cdn.intellipaat.com/wp-content/uploads/2015/08/img-1.png)

[9] Unit testing gRPC (https://stackoverflow.com/questions/37552468/how-to-unit-test-grpc-java-server-implementation-functions)

## Requirements

The requirements covered in this design specification are as follows:

1. AWS account setup and aws educate credits registration.
2. Deploying Cassandra on AWS.
3. Install java and cassandra on each EC2 instance.
4. Change the cassandra.yaml file on each ec2 to setup a cluster. change the seeds IP addresses as required.
5. Setting up initial client server in network nodes.
6. Defining initial communication API between client and server.
7. Testing node connectivity.
8. Working on replica configuration.
9. Install gRPC and setup initial communication between the client and server through message packets.
10. Dockerize the Cassandra setup for easy deployment.

## Functional Overview

***Replication Factor and Consistency levels***

The next step is to decide which replication factor and consistency level is needed. This decides the number of nodes in our cluster and if consistency or availability is needed.

For high availability the consistency level should be set to either QUORUM or LOCAL_QUORUM

For our test, Replication factor needs to be set to 5 for 5 nodes in the cluster

With RF=3 and CL=ONE we can afford to lose up to 2 nodes at the same time, if we are using CL=QUORUM, we can lose only 1 node since QUORUM requires at least 2 replicas out of 3 to be up to execute the request.

To increase our failure tolerance, we can choose RF=5, in this case with CL=ONE we can lose up to 4 nodes simultaneously without compromising our availability or up to 2 nodes simultaneously if we are using CL=QUORUM

***Eventual Consistency***

Cassandra's eventually consistent data model and node repair features ensure that the consistency of the cluster will be automatically maintained over time. So, there is a trade of between consistency and high availability.

Cassandra's architecture allows the authorized user to connect to any node in any data center and access data using the CQL language. Most requests for data, by users, at a site can be satisfied by data stored at that site (local read/write)

## Configuration/ External Interfaces

The underlying architecture is built using following:

- gRPC
- AWS
- Cassandra
- Docker

## Logging and Debug:

Cassandra uses the java standard library for logging called Log4j in its backend. To find out what is exactly happening at a particular nodetool, various logging levels can be altered for the same.

In the frontend, Cassandra uses SLF4J. The various logging levels are Debug, Trace, Error, Fatal, Warn and Info. The default logging level is Info.

To run Cassandra in the foreground with highest debugging levels the following steps should be followed:

1. Edit conf/log4j-server.properties:

   ```
   Log4j.rootLogger=DEBUG, stdout, R
   ```

2. Start the instance using f:

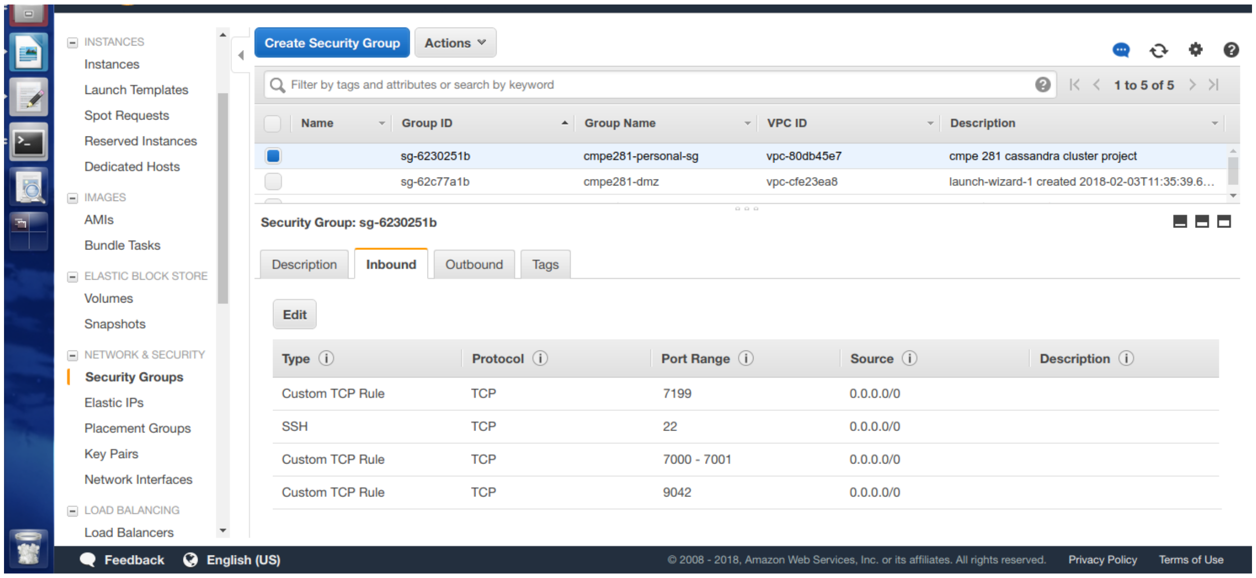
   ```
   $ bin/Cassandra -f
   ```

## Implementation

The implementation details include a task level breakdown of the underlying functional requirements.

***Cassandra Cluster Deployment***

1. Initial Configuration
2. Launch an EC2 instance of t2.micro category with security group having ports 22(SSH), 7000, 7001, 9042, 7199(Cassandra) open.



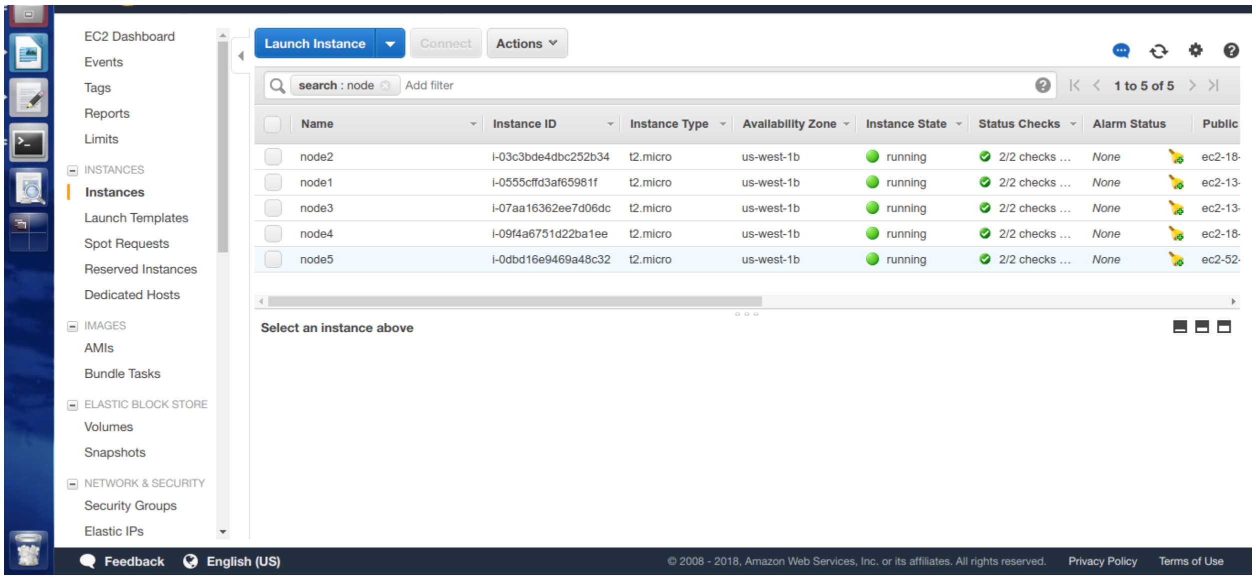
| Name | Group ID | Group Name | VPC ID | Description |
|---|---|---|---|---|
| | sg-6230251b | cmpe281-personal-sg | vpc-80db45e7 | cmpe 281 cassandra cluster project |
| | sg-62c77a1b | cmpe281-dmz | vpc-cfe23ea8 | launch-wizard-1 created 2018-02-03T11:35:39.6… |

Security Group: sg-6230251b

| Type | Protocol | Port Range | Source | Description |
|---|---|---|---|---|
| Custom TCP Rule | TCP | 7199 | 0.0.0.0/0 | |
| SSH | TCP | 22 | 0.0.0.0/0 | |
| Custom TCP Rule | TCP | 7000 - 7001 | 0.0.0.0/0 | |
| Custom TCP Rule | TCP | 9042 | 0.0.0.0/0 | |

Fig 2: Creating a VPC and security group for EC2 configuration.

3. SSH into the instance using the key selected.
4. Install Cassandra DB
5. Stop the instance and create an AMI image from the instance.
6. Use this image to launch 4 more replicated instances so that we do not have to setup everything again.



| Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks | Alarm Status | Public |
|---|---|---|---|---|---|---|---|
| node2 | i-03c3bde4dbc252b34 | t2.micro | us-west-1b | ● running | ✓ 2/2 checks ... | None | ec2-18- |
| node1 | i-0555cffd3af65981f | t2.micro | us-west-1b | ● running | ✓ 2/2 checks ... | None | ec2-13- |
| node3 | i-07aa16362ee7d06dc | t2.micro | us-west-1b | ● running | ✓ 2/2 checks ... | None | ec2-13- |
| node4 | i-09f4a6751d22ba1ee | t2.micro | us-west-1b | ● running | ✓ 2/2 checks ... | None | ec2-18- |
| node5 | i-0dbd16e9469a48c32 | t2.micro | us-west-1b | ● running | ✓ 2/2 checks ... | None | ec2-52- |

Select an instance above

© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Fig 3. All nodes (EC2 instances) up and running in AWS.

### *Cluster Configuration*

1. SSH into all nodes and change the following things in the cassandra.yaml file found in /etc/cassandra/conf/ folder
2. cluster_name: 'cmpe295A' ; seeds: ip1, ip2, ip3, ip4,ip5
3. listen_address: ip1 ; rpc_address: ip1

```
listen_address: 10.0.0.200
# listen_interface: eth0
# listen_interface_prefer_ipv6: false
```

```
listen_address: 10.0.0.200
# listen_interface: eth0
# listen_interface_prefer_ipv6: false
```

```
# ipv4. If there is only one address it will be selected regardless of ipv4/ipv6.
rpc_address: 10.0.0.200
# rpc_interface: eth1
```

4. IP addresses are the private IP addresses assigned to the ec2 (we use private ip as they do not change unless ec2 is terminated)

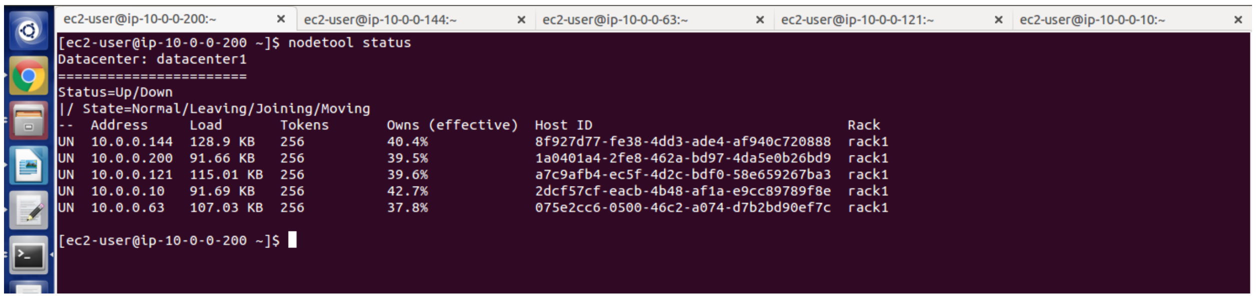
5. listen_address and rpc_address  will change to ips of the corresponding ec2.-

6. start the cassandra service in each node using the command:

    a. ‘sudo service cassandra start’

7. check the cluster status using the following command

    b. ‘nodetool status’



```
[ec2-user@ip-10-0-0-200 ~]$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID                               Rack
UN  10.0.0.144  128.9 KB   256          40.4%             8f927d77-fe38-4dd3-ade4-af940c720888  rack1
UN  10.0.0.200  91.66 KB   256          39.5%             1a0401a4-2fe8-462a-bd97-4da5e0b26bd9  rack1
UN  10.0.0.121  115.01 KB  256          39.6%             a7c9afb4-ec5f-4d2c-bdf0-58e659267ba3  rack1
UN  10.0.0.10   91.69 KB   256          42.7%             2dcf57cf-eacb-4b48-af1a-e9cc89789f8e  rack1
UN  10.0.0.63   107.03 KB  256          37.8%             075e2cc6-0500-46c2-a074-d7b2bd90ef7c  rack1

[ec2-user@ip-10-0-0-200 ~]$ 
```

Fig 4: Cassandra Cluster up and running shown through ‘nodetool status’ command.

## Testing

### General Approach

In the client server architecture, the basic testing is for checking if they can communicate with each other in a consistent manner. For testing this a general exchange of ping message is chosen.

For testing the API shared between the client and server, an API testing approach is chosen. This includes testing the specific methods defined in the protocol buffers by creating stubs.

### Unit Tests

1. Cassandra has a default configuration of CircleCI thus making unit testing easy. For this, permission for CircleCI to watch our github repository needs to be given.
2. To test gRPC communication efficiently, InProcess transport can be used. This test is single threaded and thus will be more deterministic.