Course: Distributed Systems - Semester 1 of 5785
Assignment 2

Directions

A. Due Date: 1 January 2025 at 11:55pm

B. Groups of three (3) or four (4) students may submit this assignment.

C. Code for this assignment (Ass2) must be submitted via Github using the per-assignment private repository opened for you in the organization. More details on the repository are found below.

D. There are 100 points total on this assignment.

E. What to turn in:

   (a) All source code and library files necessary to compile and run your programs

   (b) Queue server design document

   (c) Database design document

   (d) Testing document

   (e) Gradle build scripts

   (f) README.md file with:

      i. The name of the course, semester, and year

      ii. The names and IDs of all students

      iii. The task performed by each student

      iv. The number of hours worked by each student on the tasks

General Requirements

1. All of the code below must be written in Java and compilable and executable with Java 19.
2. All code must have comments - each function must have an introductory comment detailing its purpose, input parameters, and return value.

Parking System Stage 2: Distributed Data Storage

In the previous assignment we setup a parking architecture with a customer UI, Parking Enforcement Officer UI, and Municipality UI. In this assignment we will elaborate more on the data storage side. The physical architecture we will build is shown in Figure 1. Note that we have added support for a database cluster and a RabbitMQ cluster.

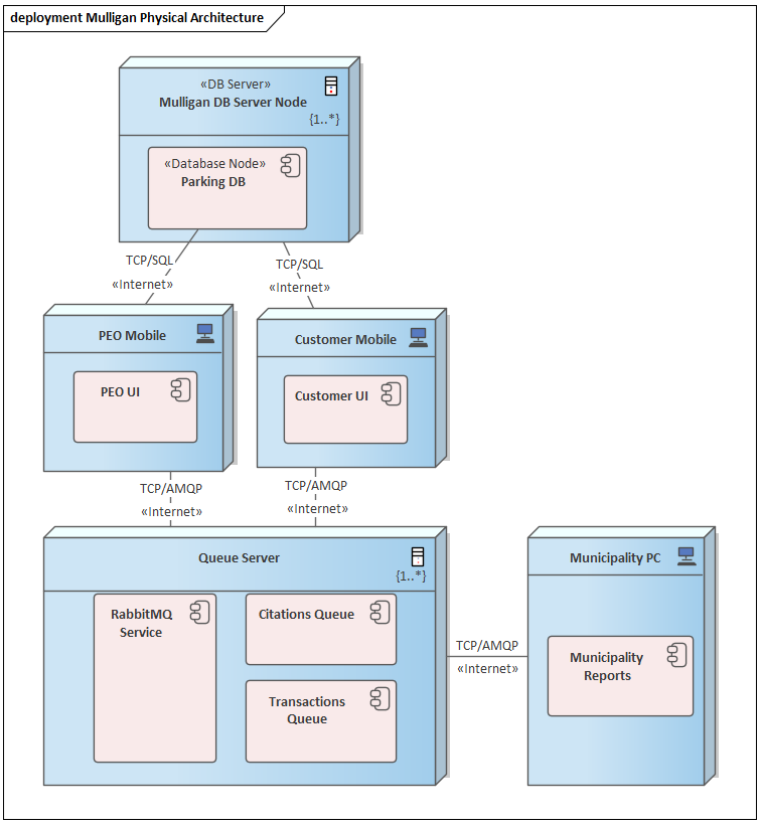

Figure 1: Mulligan distributed database physical architecture

1 Adding Clusters

The introduction of a database cluster and a RabbitMQ cluster will not affect the user interfaces that much since both systems should present a transparent interface to the cluster. Note in the logical architecture that nothing has changed (Figure 2). Since we'll be using mature distributed data storage tools contacting any node will be the same as contacting any other one, but you will need to deal with configuration and management tasks related to building the clusters.

The clusters you'll build will be small - 3 nodes each - but that is enough to allow us to get a feel for how distributed data storage works and how cluster communication is managed.

1.1 Database Cluster

The previous stage of the assignment required you to choose a database server for your backend, but didn't specify which database system to use. For this stage, you must choose a database engine that supports distribution and cluster communication.

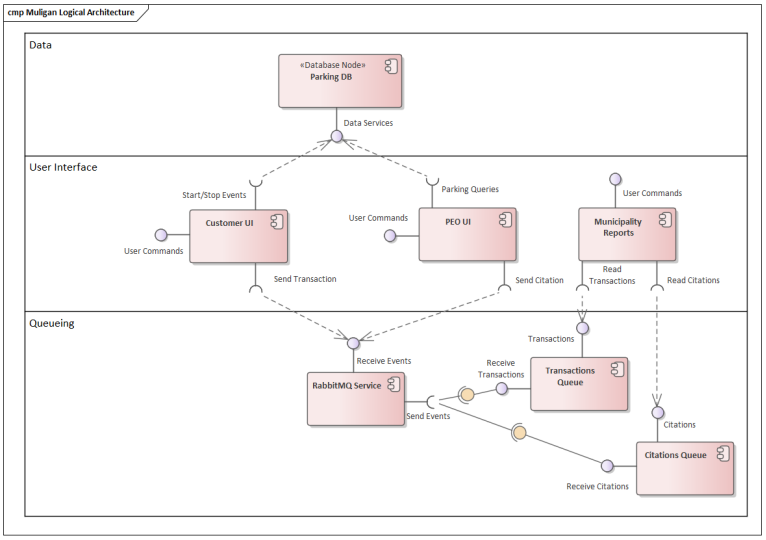

Figure 2: Mulligan distributed database logical architecture

1.2 SQL Options

If you want to use an SQL database, the following three options are recommended. Choose one.

MySQL NDB

The distributed database engine for MySQL. It's not included in all MySQL distributions, so you'll need to download special server instances from the MySQL download page. You'll need to use the NDBCLUSTER database engine and perform some configuration tasks to get three nodes up and running. The following are guides you can use to set up your cluster with MySQL. Note that all of them assume you have actual separate nodes for your running MySQL instances. You can't really run two copies of MySQL on the same computer easily unless you use containers or pods.

- How To Create a Multi-Node MySQL Cluster on Ubuntu 18.04 from Digital Ocean
- MySQL NDB Cluster installation from MySQL documentation site

Vitess

A MySQL compatible distributed database system. It has cleaner support for database sharding and distributed management. It also supports container based distribution with Kubernetes which should make it easier to run multiple nodes on a single computer for testing.

- Vitess Local Install instructions from Vitess documentation

PostgreSQL

A full featured open-source SQL database engine. The engine supports clusters, but is a bit complicated to configure. You can find many guides and help online for setting up the cluster and managing it. Some cluster setup guides use additional libraries in the process. You'll need to choose one that meets your hardware capabilities.

- High Availability, Load Balancing, and Replication from PostgreSQL documentation
- How to Setup a PostgreSQL Database Cluster a basic tutorial
- Deploying PostgreSQL for high availability with Patroni on RHEL or CentOS another tutorial that uses the Patroni high availability framework.

1.3 NoSQL Options

If you want to use a NoSQL database storage option, there are many to choose from. NoSQL databases tend to be object or document based and often have much simpler cluster setup. The following are some options you can use. For each option, I put links to a few starter guides and tutorials that may be of use.

MongoDB

A mature open-source document based database. There is a community edition available that you can download and run locally on your computer. The database supports clusters natively, so you can use it for storage. MongoDB's website offers a commercial tool called Atlas that allows you to set up cloud based clusters, but it seems to cost money.

- Convert a Standalone Self-Managed mongod to a Replica Set a tutorial from MondoDB's documentation site.
- Deploying a MongoDB Cluster with Docker from MongoDB that is container oriented.

CouchBase

Another mature open-source document based database. It's a bit more SQL-like than MongoDB and also offers a community edition alongside a paid commercial one. The community edition is limited to 3 nodes, but that is sufficient for our assignment.

- Manage Nodes and Clusters guide from Couchbase's documentation site.
- Create a Couchbase cluster using Kubernetes a tutorial that uses the enterprise edition, but ought to work for the community one too.
- Install Couchbase Server Using Docker from Couchbase's documentation site.

Apache Cassandra

A mature open-source database from Apache Software Foundation. It uses a custom query language called CQL that is reminiscent of SQL, but different. The database is sort of like SQL in that it has tables and uses a query language, but is built for massive replication and many nodes. There is no commercial version from Apache, but I'm sure you can find hosted cloud versions if you look.

- Get Started with Apache Cassandra quick start guide from Apache, but only as single node information.
- Initializing a multiple node cluster (single datacenter) from DataStax
- How To Install Cassandra and Run a Multi-Node Cluster on Ubuntu 22.04 from Digital Ocean.

You are free try other database technologies as well.

2 Replicated RabbitMQ

In addition to creating a database cluster, we also will create a cluster of 3 RabbitMQ servers. We also will use the Quorum Queues feature from RabbitMQ to make our queues more resilient to failure, making each queue have a replication factor of 3. The following guides and links may help you with the task.

- Clustering Guide from RabbitMQ's documentation
- How to Setup RabbitMQ Cluster on Ubuntu 20.04 (Tutorial) from Cloud infrastructure services.

3 Customer and PEO UI Interactions

The customer and PEO UIs must have a list of the database servers available for use and contact one as the leader for all database queries. The UIs must be flexible, able to switch from one database node to another in case of database failure. The database nodes will take care of the synchronization between them. The UIs should be able to contact any database node and receive identical data.

Both the customer and PEO UIs must also have a list of RabbitMQ nodes and attempt to contact one each time it sends a message to the broker's queues.

4 Step 2: Distributed Data Storage

There are three major tasks to be performed here. They are designed to be distributed to different people in your team. Each team must specify who took each task. Each individual will be given a personal grade for their part in the work as well as a collective team grade for the collective effort. In future iterations, team members will swap task roles, so be sure that everything is documented clearly and written cleanly.

All members of the team should help select the most appropriate database solution to use for the tool. All members of the team will need to adapt the various parts accordingly, including the proper use of the Java APIs for data storage and retrieval.

4.1 Task 1: User Interfaces

There are 3 user interfaces on the client side:

1. The Customer UI
2. The PEO UI
3. The MO UI

What to do:

1. Modify the user interfaces to work properly with the clusters as defined.
2. Introduce or adapt Java APIs for the database as appropriate.
3. Adapt the interfaces to declare and use the RabbitMQ queues and nodes.

4.2 Task 2: Queue Server

Use the existing framework for RabbitMQ, but introduce three nodes that work together in a cluster. Ensure that if any single node fails, the others can take over and not lose data.

What to do:

1. Build a 3 server cluster using RabbitMQ
2. Ensure there are two named quorum queues on the servers.

4.3 Task 3: Database and Storage

Implement the database cluster using the selected database system. The cluster must have 3 nodes and ensure data availability even under single node failure.

If you changed the database structure, adapt any queries necessary.

What to do:

1. Create and setup the database cluster
2. Create the database and schemas
3. Insert the sample data
4. Create a design document for the database, including the schemas used and important queries.

4.4 Task 4: Documentation, DevOps, Build and Test

Document all classes with JavaDoc. Write meaningful comments on all classes and public methods.

The clients and server will be stored in GitHub and written in Java. Prepare the foundation projects in the GitHub repository so that they can be built automatically and tested automatically as much as possible.

Update the testing plan to include the following cases:

1. Failure and recovery of one database node.
2. Failure and recovery of one RabbitMQ node.

5 For 3 person teams

If your team has only 3 people, distributed the documentation and DevOps task among the team members appropriately. The UI person should write the documentation and unit tests for the UIs. The queue server person should write the documentation and tests for the queue server. The database person should write the tests and documentation for the database.

6 Submission and Grading

Submit all of your work using the repository opened for you in GitHub Classroom. Submit all documents and code in the repository.

Grading:

1. 50% individual work on the task per person
2. 50% overall group grade