

Internals of Application server

Project Design

Group-3

(v.2)



Contents:

1. Introduction to the Project

2. Test Cases

- 2.1 Test cases- [used to test the team's module]
- 2.2 Test cases- [used to test overall project]

3. Solution Design Considerations

- 3.1 Design big picture
- 3.2 Overall system flow & interactions
- 3.3 Environment to be used
- 3.4 Technologies to be used
- 3.5 Approach for communication & connectivity
- 3.6 Registry & repository
- 3.7 Service & Server lifecycle
- 3.8 HA & Load Balancing
- 3.9 Interactions between modules

4. Meta Data of Modules.

5. Key Data structure

6. Interactions & Interfaces

7. Persistence

8. Low level design



1. Introduction

Definition: Platform for Smart City is a distributed platform that provides build, development and deployment functionalities. The platform will be able to deploy and run applications that can be used to manage a smart city . With this platform, we will be trying to automate the tasks that needed to be done regularly . Like applications to save power by sensing room and switching off electrical appliances, automate switching on and off street lights based on natural light also efficient gardening systems which can automatically turn on and off the sprinklers in the garden etc.

Scope: On Our platform the app developer can do app development with his own custom code updates by a set of independent services that platform provides. The platform provides various independent services like Security service (Authentication and Authorization), Build and Deployment capabilities , Logging and monitoring, Notification and Actions Service, Resource management, Scheduling service and repository to store data.

2. Test Cases

2.1 Test cases-[used to test the team's module]

Scheduler:

1. **Periodic Setting:** It will take action periodically. Its action will be mostly by considering the input periodicity or else the default mode.
2. **Multiple attempts:** It will try to make multiple attempt in case of failure or request so that it can be processed.
3. **Sensor Binding:** Scheduler has the task to bind the sensor with correct application.

Application manager:

Sensor Module:

1. Given sensor type and location will it able to filter out all valid sensor id.
2. During sensor registration will it successfully able to parse the data coming form Application manger and store the write information to registry.
3. Will it able to properly simulates the real time sensor.
4. How to control the sensor .

Communication Module:

1. If two module call it for their communication then if it is able to take data from one and send it to other.
2. How data parsing is handled in input as well as output interface.
3. How to manage sensor data of different type.
4. How to handle dynamic topic creation.
5. How to handle load and do partition on topic at runtime.

Bootstrap:

1. Is it able to understand the config file and load the system in sequence.
2. Is it able to install all basic software to make the system runnable.
3. Is it able to login to remote server to deploy the whole platform.



Logging and Monitoring:

1. The logging and monitoring service will communicate and get status of all components of the system.
2. Logging will receive status msgs of actions done at different components and write about it in the repository
3. Monitoring will check if components are running properly by sending the heartbeat message and receiving response and write about it in the repository.

Healthcare:

1. Healthcare will read from registry the status of components.
2. If component is not running properly or have some issue it will perform corresponding action
3. Also it will notify about it to SLC

Action/Notification Module:

1. It will receive data from the runtime server about the running service.
2. If any action or notification is needed to be sent it will get to know about it from data received from runtime servers
3. It will then perform the action / send notification. For performing action it will communicate with the control system which will perform required action.

Daemon Services:

1. These are the services which are required to run on a daily basis on the basis of some condition(time or event)(Fixed Rule) .
2. This module will read about services from the repository and give details to SLC.
3. The SLC will further run the demon service as required.

2.2 Test cases [used to test overall project]

Authentication and Authorisation :

1. Check Username and password

- Input : username and password
 - Output: Success or failure
 - Description : It will verify username and password from DB (repository) and return the result.
2. Check User access permission
 - Input : Username
 - Output : User's token along with list of services accessible to user
 - Description : It checks user access permission from the repository and returns a list of services accessible to the user.

Deployment Service


1. Check model is deployed/ Check Model endpoint
 - Input : Given Model API endpoint
 - Output : Got response if model is deployed and running or URL not found if model is not deployed
 - Description : It will check whether a model is deployed or not by accessing its end-point. If API is accessible we get a response. But if the model is not up and running we got an error.

Scheduling Service :

1. Check model is up between start and end time
 - Input : Start time , Endtime, Model API endpoint.
 - Output : Check Model is scheduled and running up between start and end time.
 - Description : It will check whether the model is up and running between given time. It will ping the model after every time interval.

Notification & Action Service :

1. Check Action performed or not
 - Input : Model API endpoint action code
 - Output : Check console that action performed and output seen on console or not and notification received to user registered email/mobile or not.

- 
- Description : It will check whether a user's action code is executed or not after the Model has correctly predicted output.
2. Check Notification receive or not
- Input : Model API endpoint and prediction output.
 - Output : Whether users receive notification on register mobile or email or not.
 - Description : After the model got the result then it should notify the user about it. Checking whether the user received any notification about it or not.


3. Solution Design Considerations

3.1.1 Design big picture :



3.2 Overall system flow & interactions

1. First of all, We will get the **config file** from the platform repository and start the **bootstrap program (init.py)**.
2. It will then start the **kafka** and the application repository according to the config file and also start the **mongoDB** database.
3. Then it will start the **Service lifecycle manager** which will manage the services on the platform and **Server lifecycle manager** which manages all the runtime servers info.
4. After that all the modules of the system get up and according to the dependencies.
5. Then the **Application Manager** will fetch the data from the message queue, It will make a configuration file from the extracted data and send it to the scheduler.
6. **Scheduler** will create an instance which will contain **User_id, Algo_id, Start_time, end_time, sensor_types, geolocation, action**.
7. Then Scheduler will map this algorithm id with the specific algorithm from the algorithm repository and It will schedule it accordingly.
8. This algorithm repository can be updated by the system developer.
9. Now, the scheduler will give this instance to the Service lifecycle manager and Service lifecycle manager contact the server lifecycle manager. Server lifecycle manager will select the runtime server on which this service will run.
10. **Server lifecycle manager** will check which node is having less load which is updated in the registry by monitoring service and After applying the load balancing algorithm, it will give the response back to the service lifecycle manager. The response will contain the assigned Server Ip and Server port.
11. State of each node is saved, so if any node fails then according to the last saved state it is resumed.
12. Now the Service lifecycle manager will create a service(containing the **user_id, application_id, server_ip and server_port**) and give this to the deployment manager.
13. This service information will be updated in the **registry**.
14. **Deployment manager** will bind the sensors to the algorithm instance. When the sensors are binded according to the geolocation provided by the user and then



the algorithm instance can now access the stream of data related to that sensor and then run it on the server chosen by the server runtime manager.

15. When the service is deployed, the deployment manager ping service lifecycle manager with the ack that “Service Deployed successfully”.
16. All the communication is done by kafka and the sensor will produce the data on the sensor topic.
17. Then finally that algorithm is run on that node and pushes its output on the message queue.
18. **Action server** will perform the necessary action(Email, sms, etc).
19. The **Topology Manager** manages the overall topology of the system. It is connected to the registry and it continuously checks whether a service is running or not with the help of the health checker module.
20. If the service is down then It is re-executed by the service lifecycle manager from the state which was saved in the repository.
21. **Monitoring module** will monitor all the services in the system. It will ping each and every service in the system and update its status in the registry. It will have a submodule heartbeat manager which will tell us whether it is alive or not.
22. **Logging module** will maintain all logs of the system and dump it in the repository.

How Application is deployed?

- Users will create a zip file which will include all algorithmic files.
- Users will include a config file based on the structure defined for config in the tutorial document.
- Now it will send this zip to the application manager. Application manager will first validate the user, then it will validate the format of the config file.
- If format is not fine it will report about it to the user else will solve dependencies of the algorithm and will set environment variables too.
- Now it will save the entire zip content under userID_AppName folder.

How service will run?

- ❑ **Request Manager** will receive requests from users via application interface. Now the request manager will send a token (userID, application name, service name) to the **service lifecycle manager**. Now the service lifecycle manager will send these details to the authentication server.
- ❑ Now the **authentication server** will check whether this user can use this application's service or not. For this authentication server will look into the registry and will respond back to the service lifecycle which will eventually respond back to the request manager.
- ❑ Now in case of a valid user, the request manager will create a token (containing algorithm name for that service, priority if any, dependencies if any, sensor type and location for that service , action, etc..) related to service from algorithm repository.
- ❑ Now the request manager will send these details to the scheduler. Now the scheduler will solve dependencies and will schedule the algorithm's instance accordingly. Now what to schedule along with the token will be sent to the server lifecycle.
- ❑ Now the server life cycle will assign a server , where this service's instance will run. It will add details like port number of server ,its ip address to the token. Now it will pass this token to the service lifecycle manager.
- ❑ Now the service lifecycle manager will send this token to the **deployment manager** .
- ❑ Now it's the deployment manager's job to parse the token and find details like sensor type and location. After getting these results deployment manager will ask sensor manager to provide sensor id's list , which deployment manager wants. For this deployment manager will look into the **sensor repository**.
- ❑ Deployment manager after receiving a sensor id list will bind this list with a token and will send this token to the **runtime server** (i.e the one whose details were present in the token received by the service lifecycle).
- ❑ Now the runtime server will read the token. It will find algorithm names and sensor ids for that algorithm. It will ask for data from those sensor's topic . These sensor's topic will be managed by a kafka broker.
- ❑ These topics will send data to the runtime server. Now the runtime server will run the algorithm with data as runtime argument .

- ❑ The output generated by running that algorithm will be sent to the **action server** along with a token.
- ❑ Action server will read output sent by runtime server. It will also read a token to know what action to take for that output and for that service.
- ❑ Action server will check the mapping of this action in the **action repository** to know which algorithm to run to serve this action.
- ❑ Now the action server will run an action algorithm with output received from the runtime server as runtime arguments and the user will get the result.
- ❑ There is another job which will be done by an action server. Based on the output, Action server will check for thresholds associated with service. After knowing this, if there is any discrepancy then it will inform the control unit to turn off/on some sensors.

3.3 Environment to be used:

- 64-bit OS (Linux)
- Minimum RAM requirement: 4GB
- Processor : intel Pentium i3 5th generation


3.4 Technologies to be used:

- Python framework is used to develop a platform
- NFS: Network file sharing
- Bash shell scripts for automation
- Kafka, for communication between different modules
- Sensor simulator
- MongoDB

3.5 Approach for communication & connectivity

To communicate between every module we have to define the end point of each module means what is input data for a module and what is output data for that module. Here we assume that data should be formatted in proper json format. Now once the end point has been defined we can use this information to build communication modules.

Communication module has interfaces associated for each module. Each interface is divided into two parts: input interface and output interface. Input interface is connected to



input data parser to parse the input data in appropriate format. Output interface is connected with output data parser to convert the outgoing data in suitable format.

3.6 Interaction between modules:

The interaction between modules will be done by the help of **Apache Kafka**. Kafka is an open-source stream-processing software platform, written in Scala and Java. which aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

The Kafka cluster stores streams of records in categories called topics. A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. The communication module will create Topics for each component as well as sensor and then whoever want to communicate with a component will write on the corresponding Topic and a component will receive the message sent to it by reading the data on its Topic.

Further Interaction , Communication and data exchange between different components have been described in the next sections of the document.

3.7 Registry & repository

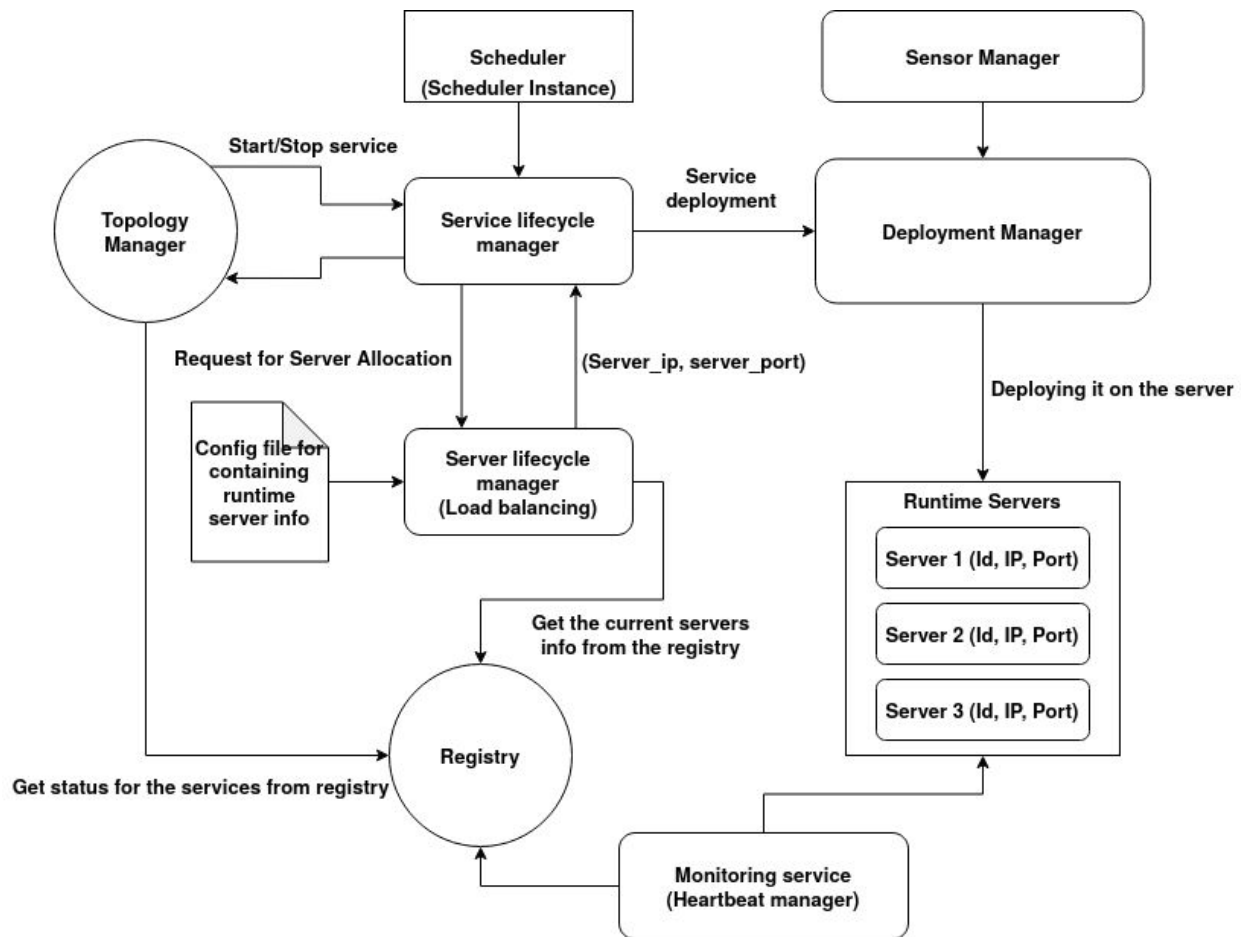
1. Registry
 - It holds dynamic content.
 - It will save the state and progress of each module running in our platform with file locations.
 - It collects data from each running module so when required to restore the state of the module.
 - For restoring, machine stats are given to the topology manager.



2. Repository

- Module needs to save dependent data, so it gets saved in its repository.
- Mainly split among users and services corresponding through that module.
- It collects data from the corresponding module and saves the config file of that module.
- When required, it sends data requested by module from its repo, it can be for authentication, type checking or formatting.
- Example: App repo, Algo repo, etc.

3.10 Service & Server lifecycle Manager



❖ 3.10.1 Service Lifecycle Manager

- Service lifecycle manager will receive details of service in json format from the scheduler. It will then communicate with the server LC manager to get the IP/Port of server on which it will run it. Topology manager will give instruction of start/stop for any service to service LC manager.
- After that it will give details to the deployment module to deploy the service on the runtime server.
- Functionalities of this module
 - Run a Service instance
 - Kill all Service instances running on single machine input (IP, username)
 - Kill a single Service instance input (ipaddress, hostname, serviceid)

❖ 3.10.2 Server Lifecycle Manager


- It will check which node is having less load which is updated in the registry by monitoring service and After applying the load balancing algorithm, it will give the response back to the service lifecycle manager. The response will contain the assigned Server Ip and Server port.
- State of each node is saved, so if any node fails then according to the last saved state it is resumed.

3.10 HA & Load Balancing

Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers. A load balancer acts as the “traffic cop” sitting in front of your servers and routing client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization and ensures about application high availability and ensures that no one server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers.

We can use any of the following load balancing algorithms:

1. **Round Robin** – Requests are distributed across the group of servers sequentially.
2. **Least Connections** – A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
3. **Least Time** – Sends requests to the server selected by a formula that combines the fastest response time and fewest active connections. Exclusive to NGINX Plus.
4. **Hash** – Distributes requests based on a key you define, such as the client IP address or the request URL. NGINX Plus can optionally apply a consistent hash to minimize redistribution of loads if the set of upstream servers changes.
5. **IP Hash** – The IP address of the client is used to determine which server receives the request.

- 
6. **Random with Two Choices** – Picks two servers at random and sends the request to the one that is selected by then applying the Least Connections algorithm (or for NGINX Plus the Least Time algorithm, if so configured).

4. Basic Entities:

Description:

1. Sensor Simulators:

- It simulates the sensor so that we can use the virtual sensor as a real sensor.

2. Gateways :

- It controls micro-services.
- It does minimal processing on input data form micro services.
- It filters out data before sending the data to the central processing unit.

3. Algorithm:

- It is the core part of the system where the logical statements are written.
- It takes input from the algorithm platform and gives the output(commands) to the algorithm platform .
- It also gives the output to the dashboard for visualization purposes.

4. Deployer:

- It helps the developers to deploy new algorithms to algo platform.

- It first pushes the source code to the algorithm repository, then it creates a run time instance of that algorithm and gives it to the algorithm platform.

5. Repository :

- It consists of the set of algorithms.
- It contains the artifacts about the algorithm.
- It contains metadata about the algorithm.

6. Binder:

- It helps developers to bind a particular algorithm to a particular input type.
- It binds algorithm to the sensor type.

7. Algorithm Scheduler :

- It chooses one algorithm from the repository based on input sensor type and gives it to the algorithm run time module.
- It can use some efficient logic to schedule the algorithm.

8. Information Modules :

- It has all the information like gateways id, microservices id, sensor name, sensor type.
- It maintains information about shared modules such as shared sensors.
- It contains each detail about every module.ex- No. of sensor, type of each sensor, etc.

9. Algorithm Platform:

- It acts as an interface between OneM2M and algorithm modules.
- It provides a base communication platform between algorithm runtime modules and other modules inside the central processing unit.

10. Run time server :

- It provides a runtime environment where an algorithm can run.

- It provides all the useful API for an algorithm to run.
- It provides a debugger also to debug programs in run time.

11. **Dashboard :**

- It provides a nice user interface for visualization of the system and data.
- It provides some control to the end user
- It also includes a data analysis part in it.
- It has a debugger to find bugs in the system.
- It also shows a warning .
- It shows information about all the sensors and how they are working.

5. MetaData of modules:

- **Metadata of Runtime Server:**

```
Metadata of RunTime Server:|
{
  {
    "Action_id":
    "User_id":
    "Output":
    "Node_id":
    "Execution Time":
    "Algo_id":
  }
}
```

- **Metadata file for Scheduler:**

```

1  {
2
3
4  {
5      "user_id":
6      "app_id":
7      "priority":
8      "request_type":
9      "days":
10     "start_time":
11     "end_time":
12     "duration":
13     "dependencies":
14
15  }
16 }

```

- Metadata for registry and Repository:

```

METADATA FOR REPOSITORY:
{
  {
    "Module Name":
    "Module ID":
    "File Name":
    "User ID":
    "Status": Save/Delete/Retrieve
    "Timestamp":
  }
}

META DATA FOR REGISTRY:
{
  {
    "Module Name":
    "Module Instance ID":
    "Status":
    "Node id":
    "Config file":
    "Sensor ID":
    "Timestamp":
  }
}

```

- Metadata for sensor registration module:

```
Sensor Registration:
{
  Type:
  Location :
  Location_type:
  IP:
  Port:
  Range:
  Frequency:
  Sign-up time:
}
```

- Metadata for sensor manager:

```
Sensor manger:
{
  Type:
  Topic Name:
  Stream_type:
  Id: [list of sensor id that satisfy the
      range and other requirement]
}
```

- Metadata for Get_sensor_topic call by Deployment service :

```
Get_sensor_topic:
{
  Topic_type:
  Topic_Name: (Dynamic)
  Data: {
    Type: []
    Id: []
  }
}
```

- Metadata for fixed rule:

```
Fixed Rule:
{
  Rule_type:
  Rule:
  {
    Strat_time:
    End_time:
    Ip:
    Port:|
    Priority:
  }
}
```

- Metadata for bootstrap configuration file:

```
config_bootstrap:
{
  Version:
  Services:
    Scheduler:
      Image:
      Ip:
      Port:
      Volume:
      Dependencies:
        App mang
    App man:
      Image:
      Ip:
      Port:
      Dependencies:
        Service Life Cycle
    Service Life Cycle:
      Image:
```

- Metadata for Service Life cycle manager:

```
{
  userd_id:'ul_user'
  app_id:'a1'
  start_time:
  end_time:
  duration:40
  dependency:
  action:'none'
  notification:{
    type:'email'
    body:'"lala"'
  }
  location:'nilgiri'
  service_path:'a/a1'
}
```

6. Key Data structures

Storage Structures

A. Sensor Information


- Sensor Type
- Sensor Location
- Sensor ID
- Gateway ID in which sensor is deployed
- Streaming rate

B. Gateway Informations

- Gateway ID
- Gateway Name
- Gateway IP
- Gateway PORT
- Sensor Lists

C. Server Information

- Server ID
- Server Instance Name

- 
- Server IP
 - Server PORT
 - Services information
 - CPU Utilization
 - RAM utilization

D. Scheduling Information

- Schedule Type
- Schedule Start Time
- Schedule Stop Time
- Model to schedule

E. User Information

- User name
- User Id
- User Password Salt
- Registered Gateways
- Models uploaded

F. Algorithm Information

- Algorithm Name
- Algorithm Id
- Sensor type to be bind

7. Interactions & Interfaces

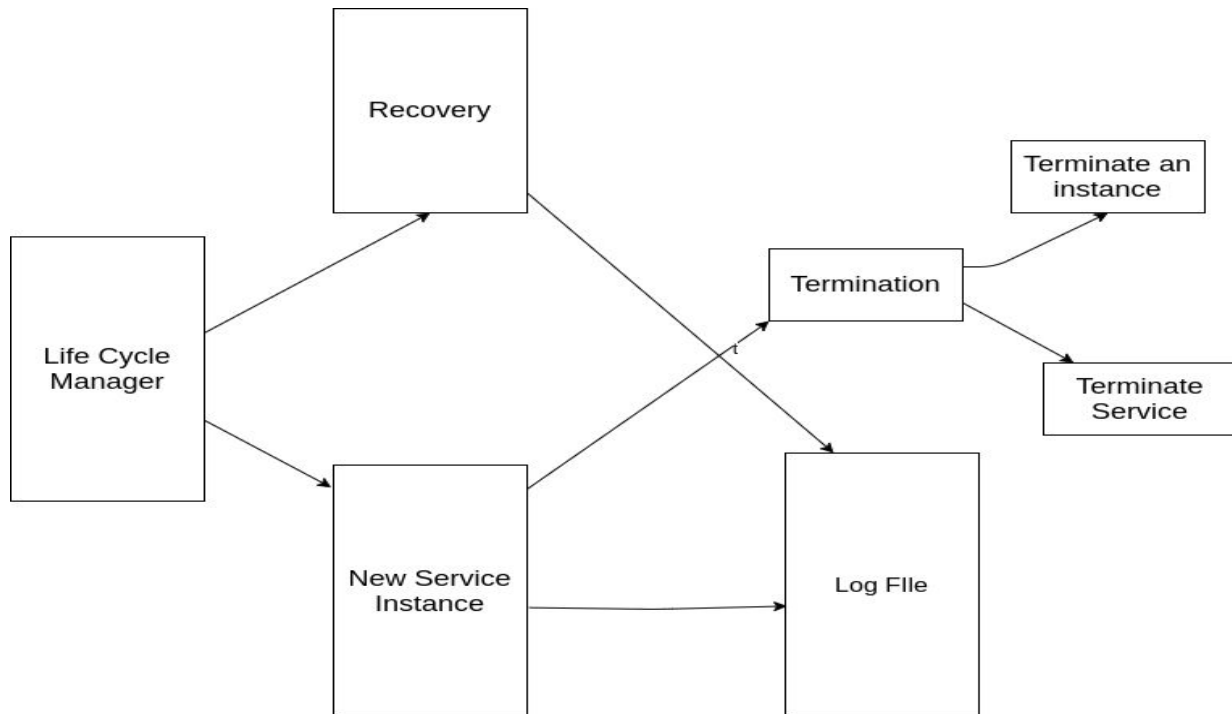
- Users of the platform will interact with the Web User Interface/Mobile App interface. User shall be able to see summarized details of the system and user specific details in a user dashboard which is going to be a landing page after the successful login of the user.
- Users shall be able to see all the deployed and to be deployed Models in the list. In the same way, an admin/special user will be able to see the up and running list of IoT devices and it's information(like IoT device Type, location of the device, capabilities etc.) in the system.

8. Persistence

- Our platform is persistent because of its high fault tolerance, whenever a node is down, then the services running on that node is shifted to some other node.
- The platform will be persistent with the use of NFS, databases and services that maintain the state of the platform.
- In case a node crashes, the system brings back it in the last state on some other node.
- In case any system or application service or running model crashes then it will be brought back by healthCheck service.

9. Low Level Design :

1.Life Cycle of module:




2. List the sub modules:

(a) Application Manager:

It manages all incoming requests from user through various apps, also ensuring user is valid and is sending data in described platform format. Further sending it to scheduler and service life cycle manager.

Problem Space:



It has to resolve all dependency regarding application config validity and user authentication. Firstly Application manager will authenticate the user , and check if it is a valid user or not. If valid user, then check structure of config JSON file, if correct structure, store it in App repo . If new service, then give it to scheduler for allocation of new node. If existing service then to start/stop, give to service lifecycle manager.

(b) Scheduler:

Scheduling component is a microservice based architecture of a platform that coordinates with all other subsystems and different services. It will also manage scheduling of models on gateway or server with different scheduling policy.

Problem Space:


This platform will provide a set of independent services which will take input from the application manager module and deliver it to the topological balancer. This component of the project will map the correct algorithm with the required service.

(c) Topological Manager :

Topological Manager contains run-time information of services. Load details regarding services and threshold info from Repository (Config). Get information Machine stats info from Registry. Decides how many services to run, where to run services, what services need to run. Request Service Manager to start/stop services. Update load balancer about Service instances for request routing

(d) Communication Platform with simulator:

Communication component is a microservice based architecture of a platform that provides an interface to do communication between all other subsystems. It will also provide some basic input output interface for end users. The Communication module is made with a message queue which is segregated into different channels(topic). In each topic the structure of message is predefined.



The simulator module will provide the sensor data by running some algorithms. It is basically software simulation model of sensors. The main aim of this module is to provide sensor data in raw format.

Problem Space:

The platform has many different modules for example, user interface, output interface, scheduler etc. The aim of the communication module is to provide a basic platform for intercommunication between all subsystems. It takes data from subsystems and pushes the data into the appropriate topic in the message queue, then it retrieves the data from the message queue and gives it to the appropriate module. The main aim of the sensor simulator module is to simulate the virtual sensor.

3. Brief overview of each sub module:

1. Application Manager:

It manages all incoming requests from user through various apps, also ensuring user is valid and is sending data in described platform format. Further sending it to scheduler and service life cycle manager. All new app requests pass through this module, so it needs to be robust.

2. Authentication Server:

The main work of the Authentication service is to check the data about users from the database for verification of users. Also it will be required to check if the service asked by a user is within his access rights so that a user with lower level access cannot use or change data or details of service that can be changed/ managed by a user of high access level.

3. Scheduler

Scheduling component is a microservice based architecture of a platform that coordinates with all other subsystems and different services. It will also manage scheduling of models on gateway or server with different scheduling policy.

4. Server Lifecycle Manager

Server Life cycle manager manages the life cycle of the running server. It has a list of available ip and port and also it contains a Load balancer as a sub component. Whenever a service life cycle manager communicates with Server LC manager, it sends an ip and port of node that have a minimum load to service life cycle. Server lifecycle manager will get the server's status from the registry. Monitoring service which has the heartbeat manager (to get the information about the available servers whether alive or not), It will update the registry.

When the server lifecycle manager gets the stats then it checks the load on each run time server and then decides on server load is minimum.

It then sends that runtime server's IP and port to the service lifecycle manager.

5. Service Lifecycle manager


Service lifecycle manager will receive details of service in json format from the scheduler. It will then communicate with the server LC manager to get the IP/Port of server on which it will run it. Topology manager will give instruction of start/stop for any service to service LC manager. After that it will give details to the deployment module to deploy the service on the runtime server.

6. Topology Manager

Topological Manager contains run-time information of services. Load details regarding services and threshold info from Repository (Config). Get information Machine stats info from Registry. Decides how many services to run, where to run services, what services need to run. Request Service Manager to start/stop services. Update load balancer about Service instances for request routing

7. Health Manager

Health managers main task is to read the registry and if found some inconsistency in data then tell about it to the SLC. In the registry data will be written by logging



and monitoring systems. Logging system will write logs of all systems, while the monitoring system will send a type of ping or heartbeat message to all components and write about responses received in the repository . If response is not received or bad in some kind then Health manager on reading about it will tell SLC which will perform the required action.

8. Logging Service & Monitoring Service:

The Task of logging and Monitoring service are kind of same in sense that they connect with all components and communicate with them. The main difference between them is that logging service logs of some event that had occurred on components and writes details about the event in the registry while Monitoring services will just keep on pinging the components periodically in form of heartbeat message and receive response so that it can know that the component is working properly and write about it in repository . which can be further conveyed to SLC.


9. Deployment Manager

Deployment manager's task is to bind the sensor to service. It will send location of sensors to sensor manager and get the topic and id for respective sensors for which it. After getting details of the sensor it will bind action with the service. Now with all the details of a run time server it will deploy the wrapped service to the run time server.

10. Sensor Manager

Sensor manager working is to bind sensor data with algorithms running at that instance and sending the data as per the configuration. Sensor manager manages data binding of sensor data to the algorithm without fault. . It needs to get data and send it at a suitable rate. Sensor manager will have access to all the details of sensors, using this it will dynamically create a temporary process and Kafka topic for making the binding algorithm and sensor data.

11. Run time Server



Run time server is a distributed set of running machines. Every node has its ip and port by which it is uniquely identified. A service is deployed on any one machine and its output given to the action manager. Each machine also sends its stats and load to a monitoring service for fault tolerance.

12. Action Manager

13. Communication Module

Communication module will produce the interface through which any module can communicate with each other. For ex- if deployment service want to communicate with runtime server then deployment service will call the input interface of communication module . Communication module then pass the data via kafka channel to the runtime server.

4. Interactions between sub modules:

(a) Interaction between Service Module and Topological Module:


Service module will take input from application manager and after selection of algorithm instance will send it to topological module for balancing load and process request according to the unit load.

(b) Interaction between communication module and application manager:

Application manager will fetch the data from the message queue and do the necessary action like-

- It will fetch the data from the sensor topic and process the data to find out which algorithm to run .
- It will fetch the data from command topic and give the output to the control system.

(c) Interaction between communication module and sensor simulator:

- 
- Sensor consumer will push its data to the message queue in the sensor topic.

Yes you can say that it is not transparent enough to please one, and the administration has its own logic for it. But it is how it is, you and I can't change it so there is no point discussing it.