IAS - Group - 7 (Team - 3)

Design Document

Yagyesh Purohit (2020201013)
Punit Sharma (2020201029)
Aditya Rathi (2020201041)

Nodes and Node Manager (VM's)

This module is used to manage and allocate new nodes (virtual machines) as and when required. It provides a node details along with node id, when a node request is made.If it is unable to satisfy the request a new node is created. It also manages the status of nodes, i.e. active, free or down.

Lifecycle of the module

- As the platform initializes the node manager module is started along with say k number of nodes (Virtual Machines).
- As the request of a node arises from the deployer node info is provided to the deployer.
- The node is then used to run the user application.
- The nodes are continuously monitored for failures using the HeartBeat client.
- Node manager and nodes exist till the Platform is turned off.

Sub Modules

- **Nodes** - Nodes are the entities where execution of an application instance or a platform component takes place. This may be a physical server or a virtual machine. The nodes may also communicate with the sensor manager for sending/receiving data to/from sensor(s).

- **Node Manager** - Node Manager module is responsible for managing the node information and providing the deployer a new node if needed.

Functionality of the module

- Providing a new virtual machine/node to the deployer when required.
- Providing a list of active virtual machines/nodes if required.
- Maintaining the node registry.
- Shutdown a virtual machine.

Diagram

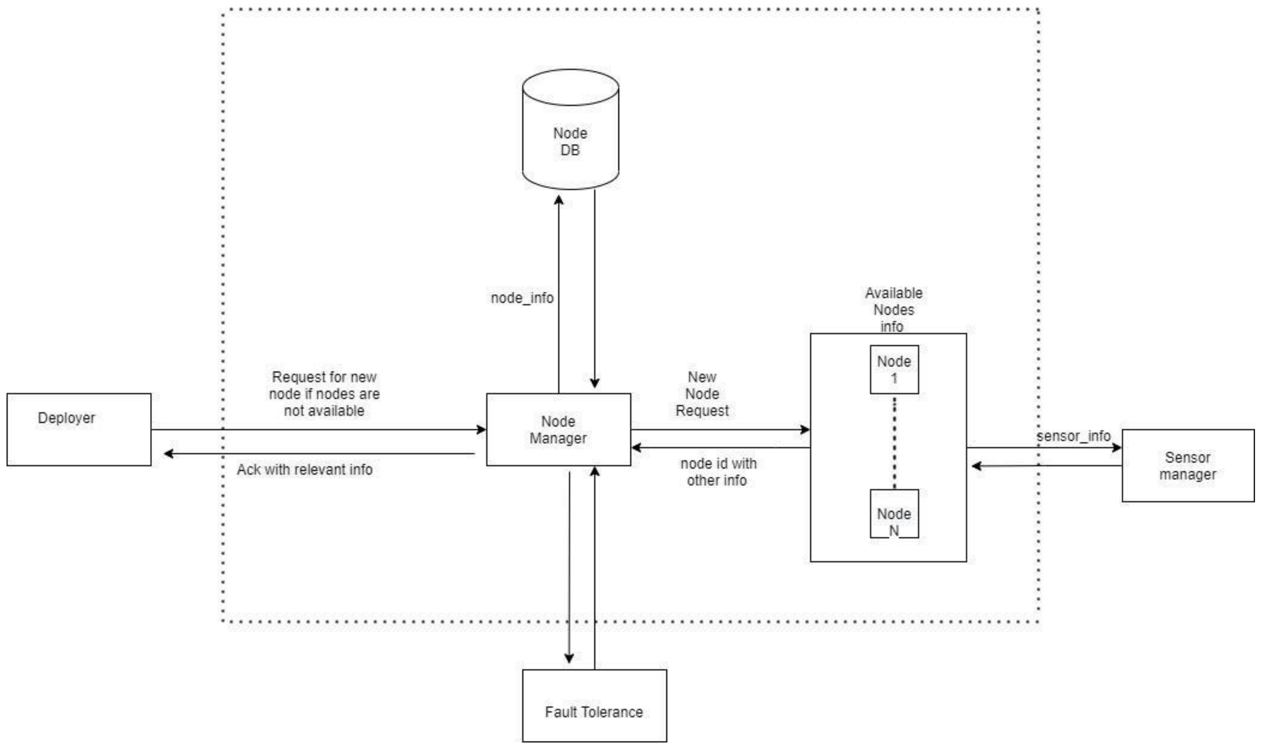


Figure 1: Nodes and Node Manager module

Interaction of Node Manager with other Modules

- **Sensor Manager**
  - An application instance running on a particular node may wish to communicate with the sensor manager to send/receive data stream to/from a particular sensor or multiple sensors.

- **Monitoring and Fault tolerance**
  - Sends the heartbeat message to the Monitoring and Fault Tolerance module. Periodic monitoring is done and their proper functioning is checked. If any discrepancy or fault is detected in the working of these modules, the fault tolerance module's instance tolerance module.
  - Heartbeat format
    - Message containing **"HeartBeat"**

Diagram:

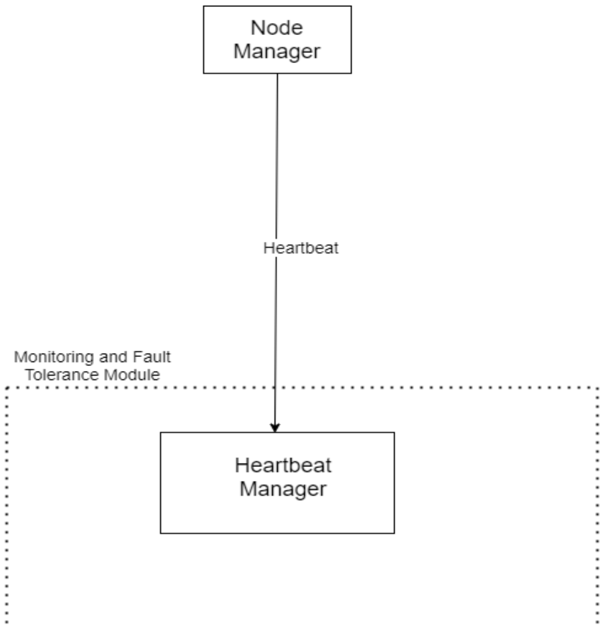


Figure: Interaction of Node Manager and Heartbeat Manager

- **Deployer**
  - The deployer would request the node manager for a new node in case the load on present active nodes increases.
  - **APIs**
    - **get_list_of_active_node**
      - Return the list of active node containing <ip,port> details
        - 127.0.0.1 8082:127.0.0.1 8085:127.0.0.1 8086
    - **create_new_node**
      - Creates a new node and sends its ip and port to the deployer.
        - 127.0.0.1 8089

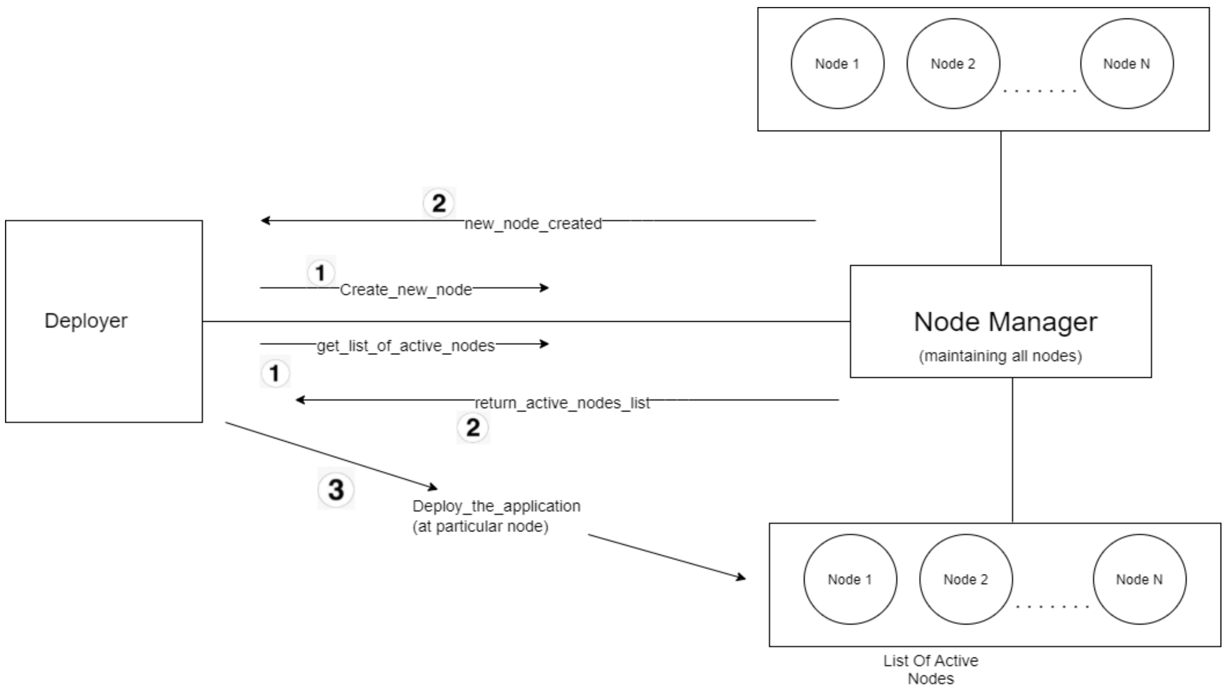


Figure 2: Interaction between Deployer and Node Manager

Test cases

Input: Deployer asking for execution of an algorithm when the available nodes are occupied.
Output: Return the new node id to the deployer
Input: Application instance requiring data from 4 sensors.
Output: Pass the application id and corresponding sensor ids from the node (on which that application instance is running) to the sensor manager. Then receive the data from the sensor manager and pass it to that application.

Monitoring and Fault Tolerance

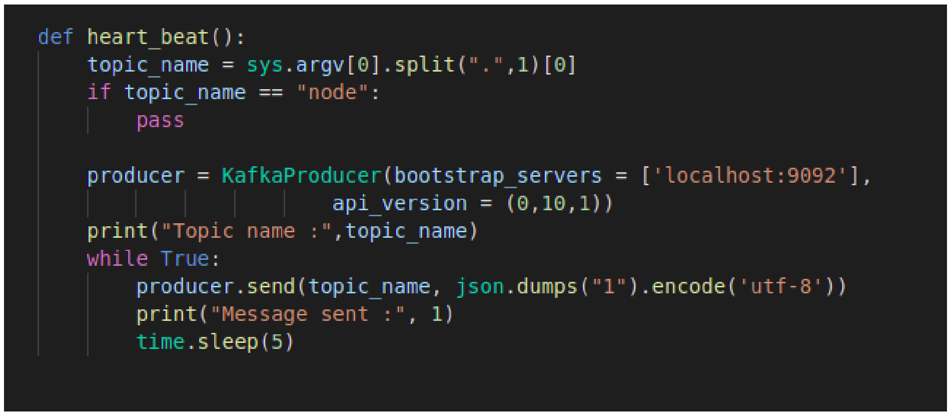
This module continuously monitors the components for failures and allows the system to operate properly, even in case of failure of one or more of its components.
It monitors the platform modules/application instances and nodes using heart beat messages. It continuously checks if the module/application and node is working properly or node. If it finds that some node is not working properly, then it initializes a new node along with the applications running on it, else it reinitializes the application on the same node.

Lifecycle of the module

- As the platform initializes the fault tolerance module is started.
- As a new application is started it is registered with the fault tolerance module. Similarly new nodes are also registered with it.
- The module continuously monitors other applications and nodes for failures using the HeartBeat client.
- Fault tolerance module exists till the Platform is turned off.

Sub Modules

- **Heartbeat Client**
  A heartbeat client is an api which every platform module/service (scheduler, deployer, node manager etc.) needs to run so that the monitoring and fault tolerance service can identify the status of these services on a continuous basis, and restart the service if it is down for some reason.
  API provided
  - **heart_beat()**
    - Sends a heart beat to the Monitoring Module.
    - Format
      - node : node-<ip>_<port>
      - application : application-<app_instance_id>
      - services : service-<service_name>

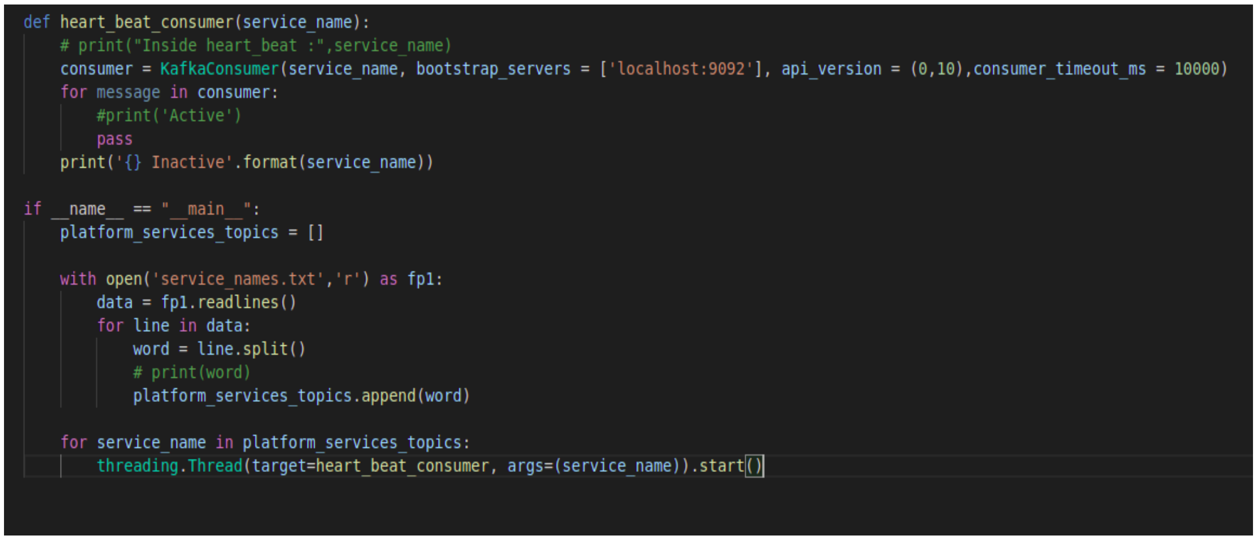


```
def heart_beat():
    topic_name = sys.argv[0].split(".",1)[0]
    if topic_name == "node":
        pass

    producer = KafkaProducer(bootstrap_servers = ['localhost:9092'],
                             api_version = (0,10,1))
    print("Topic name :",topic_name)
    while True:
        producer.send(topic_name, json.dumps("1").encode('utf-8'))
        print("Message sent :", 1)
        time.sleep(5)
```

Figure: HeartBeat Client

- **Heartbeat Manager**
  A heartbeat manager is responsible for monitoring the status of each service. If any service stops abruptly, an error message is generated, telling the heartbeat manager which service is stopped.



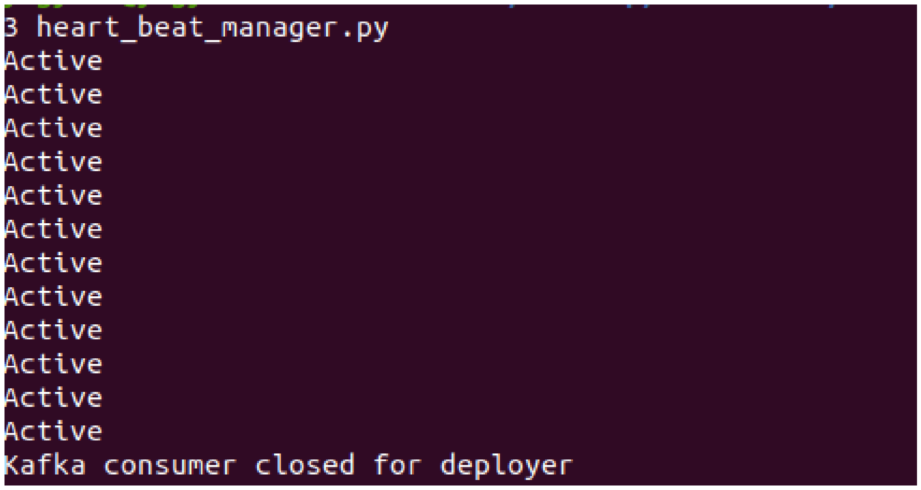
```python
def heart_beat_consumer(service_name):
    # print("Inside heart_beat :",service_name)
    consumer = KafkaConsumer(service_name, bootstrap_servers = ['localhost:9092'], api_version = (0,10),consumer_timeout_ms = 10000)
    for message in consumer:
        #print('Active')
        pass
    print('{} Inactive'.format(service_name))

if __name__ == "__main__":
    platform_services_topics = []

    with open('service_names.txt','r') as fp1:
        data = fp1.readlines()
        for line in data:
            word = line.split()
            # print(word)
            platform_services_topics.append(word)

    for service_name in platform_services_topics:
        threading.Thread(target=heart_beat_consumer, args=(service_name)).start()
```

Figure: Heartbeat manager



```
3 heart_beat_manager.py
Active
Active
Active
Active
Active
Active
Active
Active
Active
Active
Active
Active
Kafka consumer closed for deployer
```

Figure : use of heartbeat message

- **Fault Tolerance** - When it receives information of a module/node being down, it reinitializes it using its init configuration details.

Functionality of the module

- **Platform** - To check whether all the components (like scheduler, deployer, application manager etc) of the platform are working properly, and in case of failure reinitialize the component.

- **Application** - To check if the application instances are working properly or not. If they are not working then re-initialize another app instance. If some active node gets down, then it reinitializes all the applications/modules running on the node on some other node

Diagram:

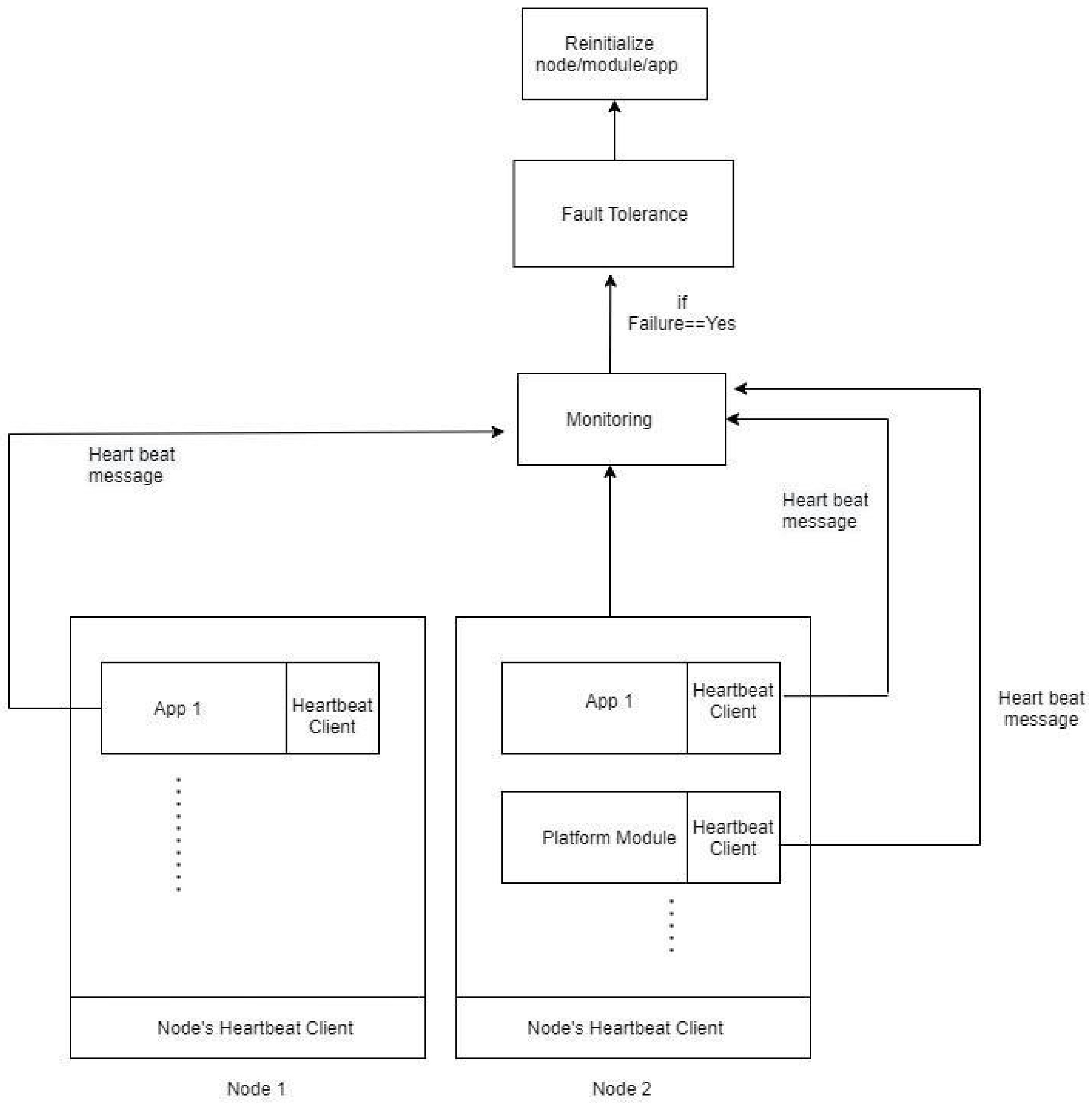


Figure 3: Monitoring and Fault Tolerance

Interaction between monitoring sub-module and fault tolerance sub-module

- The monitoring module will continuously monitor the heart-beats of all the currently running nodes. If a node is down, the monitoring module would not receive the heart-beat of that node's heart-beat client. In that case, the monitoring module would call the fault tolerance module and fault tolerance module will reinitialize that node.

Interaction of Monitoring and Fault Tolerance with other Modules

The monitoring and fault tolerance module interacts with the following modules:

- Scheduler
- Sensor Manager
- Load balancer
- Deployer
- Application Manager
- Node Manager
- Nodes and application instances

For all the aforementioned modules, periodic monitoring for their proper functioning will be done. If any discrepancy or fault is detected in the working of these modules, the fault tolerance module's instance related to that module gets triggered, and an alternate node is allocated to run that module.

Test cases:

- **Input** : node failure (did not received node's heartbeat message)
  Output : A new node with all the previous applications running.
- **Input** : Application instance interrupted / stopped in between.
  Output : Reinitialize the application running on a different node(chosen after load balancing).

Communication protocol:

- **Kafka**
  - **For monitoring service**
    - For every platform service, a KAFKA topic has been created, which logs the periodic 'active status' messages sent by different services into their corresponding topics.
    - Whenever any service is abruptly stopped, the monitoring service waits for a threshold time (10 seconds) and then displays a message that the service has been stopped.
  - **For sensor data binding**
    - There exists a topic for each sensor in kafka. Once the sensor binding is done with its uid, the data is read from the partition as required by application and send to the application

Environment used

- OS: Linux
- Docker - Environment for independent execution. Docker enables isolation of applications from each other. Each of the platform services as well as application instances are run within separate docker containers. Docker containers can also be run from remote machines using ssh, thus enabling distributed application feature.

Technologies used

- Python - Scripting language
- Kafka - Communication channel
- Virtual Machines(MS Azure) - For distributed architecture

Registry & repository

- **Node Database**

  Registry will be used to store several run-time related information for applications and platform modules.
    - Number of nodes available
    - Unoccupied node id list
    - Per node information
        - Node id
        - Node status - occupied/unoccupied.
        - Type of service running - platform module/ application.
        - Applications
            - Application id
            - Application status
            - Sensor ids
                - sensor id1
                - sensor id2

- **Monitoring Database**

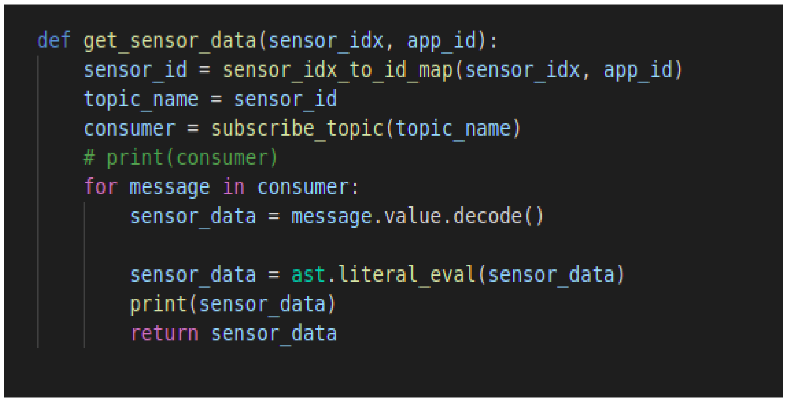
  The registry contains the following information:
    - Node info
        - Node id
        - Node status - 0 (failure), 1 (running)
        - Node failure count
        - Application details
            - App/module id
            - App status - 0 (failure), 1 (running).
            - Failure count

Sensor Data Binding

An interface named interface_to_get_data is created, consisting of following functions for fetching real time sensor data:

get_sensor_data()

- It takes the **sensor_idx and app_id** as input and resolves the sensor_idx with the physical sensor id.
- Then it fetches **the current data** from the kafka queue and sends it to the application.



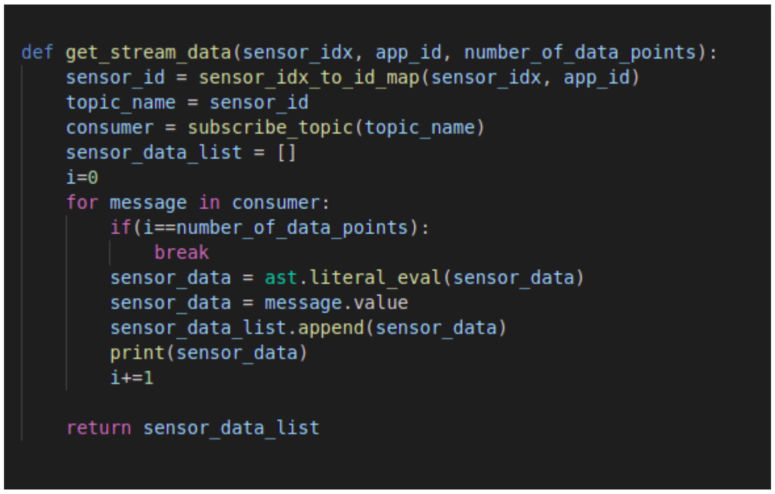
```python
def get_sensor_data(sensor_idx, app_id):
    sensor_id = sensor_idx_to_id_map(sensor_idx, app_id)
    topic_name = sensor_id
    consumer = subscribe_topic(topic_name)
    # print(consumer)
    for message in consumer:
        sensor_data = message.value.decode()

        sensor_data = ast.literal_eval(sensor_data)
        print(sensor_data)
        return sensor_data
```

Figure: To get sensor data

get_stream_data()

- It takes the **sensor_idx and app_id and the number of data instances** as input and resolves the sensor_idx with the physical sensor id.
- Then it fetches the **said number of data** from the kafka queue and sends it to the application in the form of a **list**.



```python
def get_stream_data(sensor_idx, app_id, number_of_data_points):
    sensor_id = sensor_idx_to_id_map(sensor_idx, app_id)
    topic_name = sensor_id
    consumer = subscribe_topic(topic_name)
    sensor_data_list = []
    i=0
    for message in consumer:
        if(i==number_of_data_points):
            break
        sensor_data = ast.literal_eval(sensor_data)
        sensor_data = message.value
        sensor_data_list.append(sensor_data)
        print(sensor_data)
        i+=1

    return sensor_data_list
```

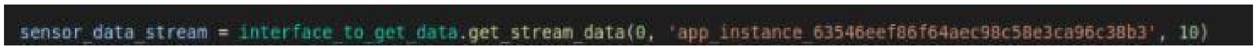
Figure: To get Data Stream

set_controller_data()

- This API is designed to update the state of a particular controller that is bound to some application instance.

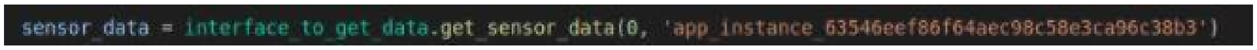
API usage

- get_sensor_data()



```
sensor_data_stream = interface_to_get_data.get_stream_data(0, 'app_instance_63546eef86f64aec98c58e3ca96c38b3', 10)
```

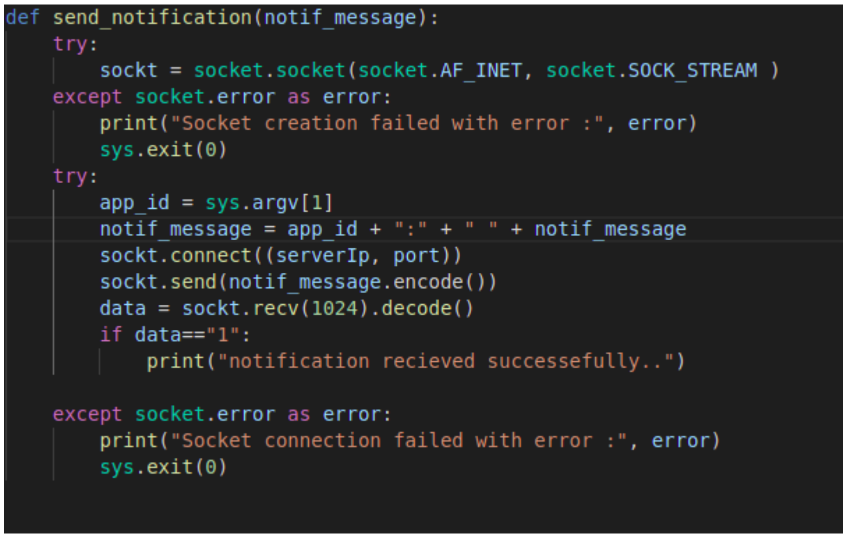
- get_stream_data()



```
sensor_data = interface_to_get_data.get_sensor_data(0, 'app_instance_63546eef86f64aec98c58e3ca96c38b3')
```

Notification

An api for handling notification requests is created. It contains the definition of **send_notification( ) function**, which facilitates the notifications to be displayed to the user.



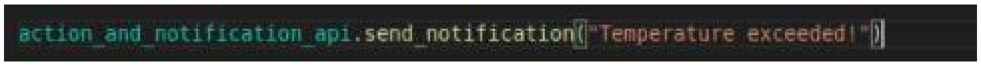
```
def send_notification(notif_message):
    try:
        sockt = socket.socket(socket.AF_INET, socket.SOCK_STREAM )
    except socket.error as error:
        print("Socket creation failed with error :", error)
        sys.exit(0)
    try:
        app_id = sys.argv[1]
        notif_message = app_id + ":" + " " + notif_message
        sockt.connect((serverIp, port))
        sockt.send(notif_message.encode())
        data = sockt.recv(1024).decode()
        if data=="1":
            print("notification recieved successefully..")

    except socket.error as error:
        print("Socket connection failed with error :", error)
        sys.exit(0)
```

Figure: To Send Notification

API

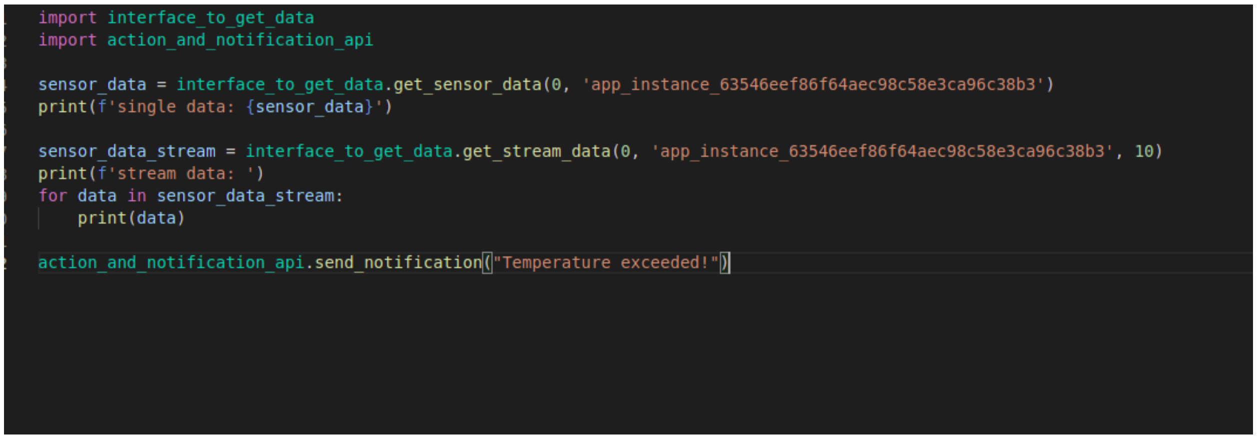
- send_notification()



```
action_and_notification_api.send_notification("Temperature exceeded!")
```

Usage of API's in the Application Code

The figure below depicts the usage of the above API's in the application.



```python
import interface_to_get_data
import action_and_notification_api

sensor_data = interface_to_get_data.get_sensor_data(0, 'app_instance_63546eef86f64aec98c58e3ca96c38b3')
print(f'single data: {sensor_data}')

sensor_data_stream = interface_to_get_data.get_stream_data(0, 'app_instance_63546eef86f64aec98c58e3ca96c38b3', 10)
print(f'stream data: ')
for data in sensor_data_stream:
    print(data)

action_and_notification_api.send_notification("Temperature exceeded!")
```

Figure: Sensor Binding and Notification