IoT Thermostat System Technical Documentation

Mentor: R Badrinath

Krish Dave (**IMT2022043**)

Vaibhav Mittal (**IMT2022126**)

Contents

1 Note:

This is our report describing our implementation of IOT-based thermostat from scratch using Java, MQTT and Python. To run the project please access the github repository which has a readme which describes what all libraries to install and how to run our project.

We have also made a ppt which has information about our motivation to do this research, our research in the field of IOT and also the things we learn during this semester in IOT. We have also included the Future work and the challenges faced by us while implementing our approach.

2 Abstract

This document details an intelligent IoT thermostat system implementing:

- MQTT-based pub/sub architecture
- Drools rule engine integration
- W3C Web of Things (WoT) Thing Description
- RDF-based semantic data storage
- Secure cloud communication

3 System Features

3.1 Core Features

- Real-time temperature monitoring (18°C - 25°C range)
- Rule-based heater control using Drools
- MQTT over TLS secured communication
- Semantic data storage in Turtle (TTL) format
- Automatic reconnection handling
- Multi-threaded architecture

3.2 Advanced Features

- WoT Thing Description compliance
- Quality of Service (QoS 1) message delivery
- Connection health monitoring
- JSON payload validation
- Error handling and alerting system

4 System Architecture

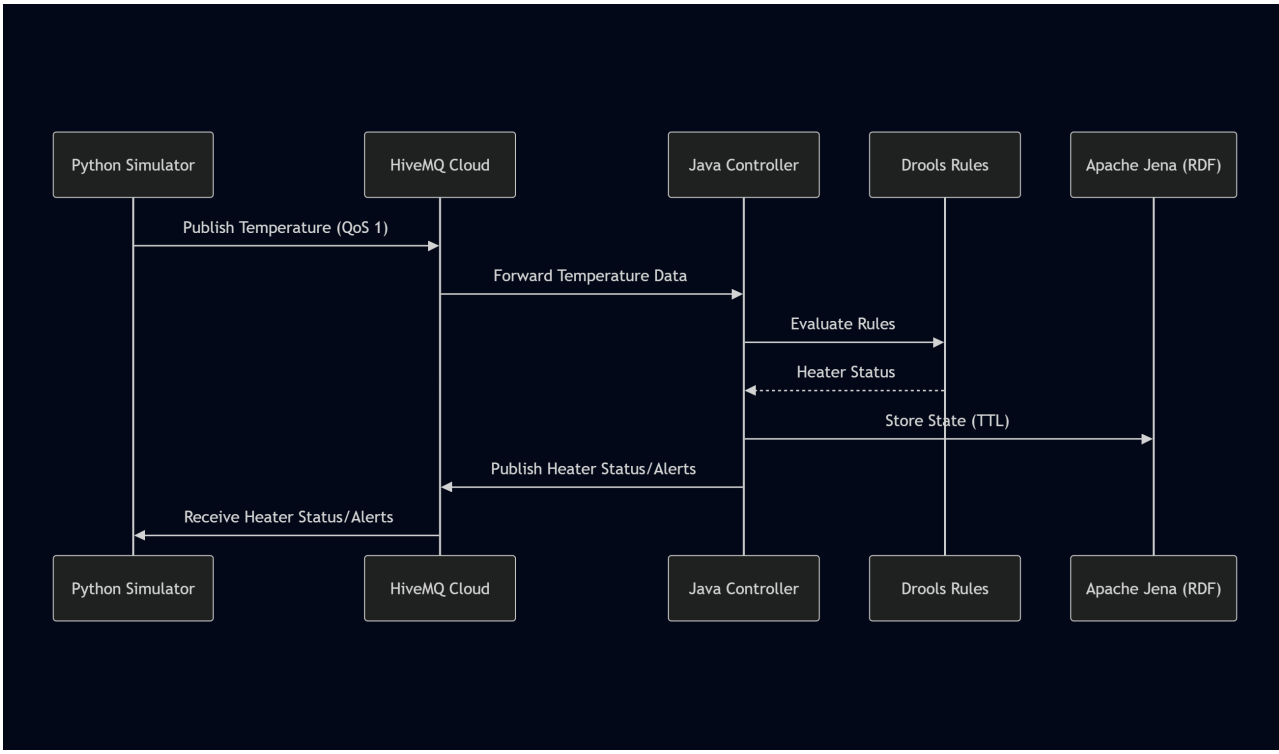

Figure 1: System Architecture Diagram

Components:

- **Thermostat Simulator (Python)**: Generates synthetic temperature data
- **HiveMQ Cloud Broker**: Managed MQTT message broker
- **Thermostat Controller (Java)**: Core business logic component
- **Knowledge Base**: Drools rule engine + Apache Jena triple store

5 Workflow

1. A Python simulator generates temperature readings in the range 18–25°C (±0.5°C).
2. The Python client publishes these readings as JSON payloads to the MQTT topic `thermostat/temperature` with QoS 1.

3. The Java-based system (MQTT client) securely subscribes to the `thermostat/temperature` topic using SSL.
4. Upon receiving a temperature message, the Java system:
    - Parses the JSON content to extract temperature and timestamp.
    - Validates the timestamp to ensure the reading is recent (not stale).
5. A Jena RDF model is constructed or updated with:
    - Device identifier
    - Current temperature
    - Timestamp
6. The RDF model is persisted to an RDF triple store for semantic storage and querying.
7. A 'Thermostat' Java object is populated with the latest data and inserted into a Drools session.
8. The Drools rule engine evaluates the inserted fact(s) against predefined rule.
9. If any rule is triggered:
    - The heater status is updated (ON/OFF).
    - The updated status is added to the RDF model.
    - The RDF model is re-persisted to reflect the new state.
10. The updated heater status is published to the MQTT topic `thermostat/status` for external consumers.
11. The Python client subscribes to the `thermostat/status` topic to receive updates about the heater's state (ON/OFF).
12. Upon receiving the updated heater status:
    - The Python client adjusts its temperature simulation based on the heater's status:
        - If the heater is ON, the Python simulator may simulate a gradual increase in temperature.
        - If the heater is OFF, the Python simulator may simulate a gradual decrease in temperature.
13. The Python client then publishes the updated temperature readings back to the `thermostat/temperature` topic to trigger the next cycle.

6 Java Client Side

This section provides a comprehensive explanation of a Java-based IoT thermostat system that integrates the following technologies:

- MQTT for messaging and real-time data exchange.
- WoT (Web of Things) Thing Description for semantic interoperability.
- Apache Jena for RDF-based semantic knowledge representation and persistence.
- Drools for rule-based decision making and automation.

The system listens to temperature data from sensors and controls the thermostat (e.g., turning it on or off) based on semantic rules and reasoning. Device states and environmental data are also persisted in an RDF triple store for semantic querying.

6.1 Secure Communication via SSL

- The system sets up SSL using Java system properties for keystores and truststores.
- SSL encrypts all MQTT traffic, ensuring secure data transmission across networks.
- This is essential for preventing man-in-the-middle attacks and protecting sensor and actuator data.

6.2 Semantic Modeling with Apache Jena

- Apache Jena is used to construct an RDF model of the thermostat.
- Triples are used to semantically describe the device, e.g., its identifier, type, current temperature, and operational state.
- For example:
  - `<thermostat1> <hasTemperature> "30"`
  - `<thermostat1> <hasStatus> "ON"`
- This structured representation enables semantic reasoning and SPARQL querying across device metadata.

6.3 Storing into RDF Triple Store

- Once the RDF model is built using Jena, it is persisted into a triple store, enabling long-term storage and advanced querying.
- The triple store allows:
  - Tracking historical data, such as temperature trends.
  - Semantic querying using SPARQL endpoints.
  - Integration with knowledge graphs or other linked datasets.
- The system can use TDB or Fuseki for storing and exposing RDF graphs.

6.4 Web of Things Thing Description (WoT-TD)

- The thermostat is described using a Thing Description (TD), conforming to WoT standards in JSON-LD.
- The TD describes:
  - **Properties:** e.g., `currentTemperature`, `status`.
  - **Actions:** e.g., `turnOn`, `turnOff`.
  - **Security:** e.g., token-based or basic security metadata.
- This enables automatic discovery and interoperability with third-party IoT platforms.

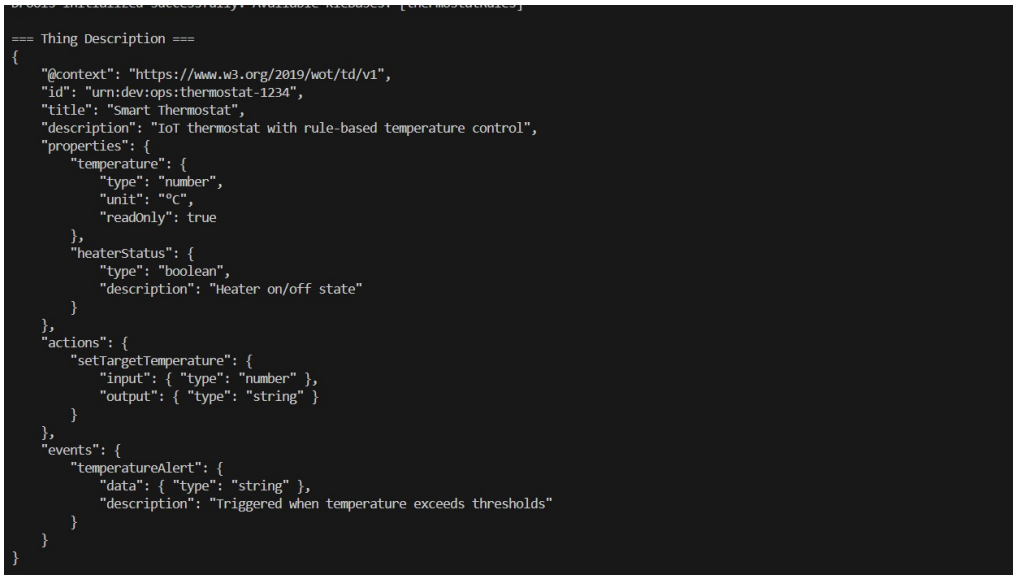

Figure 2: Sample Thing Description created

6.5 MQTT Communication

- The system uses an MQTT client to:
  1. Subscribe to a topic (e.g., `temp/sensor`) to receive temperature data.
  2. Publish control commands (e.g., `ON` or `OFF`) to an actuator topic (e.g., `thermostat/control`).
- Each published message reflects the decision made by the rule engine.

6.6 Rule-Based Automation with Drools

- Drools is used to define conditions under which the thermostat should change state.
- The system workflow includes:
  1. Creating a fact object from sensor data.
  2. Inserting the fact into a Drools session.
  3. Drools fires matching rules and outputs a control action.
- After a rule is triggered:
  - The device's new status (e.g., `ON`) is updated in the RDF model.
  - This status is stored into the RDF triple store.
  - The new status is also published to the MQTT broker (e.g., `thermostat/status` topic).

6.7 Integration Workflow

1. The MQTT client connects securely using SSL.
2. The WoT TD is created to describe the thermostat capabilities.
3. Semantic metadata is initialized and stored using Apache Jena.
4. Incoming sensor data is received via MQTT and parsed.
5. Data is passed into Drools as facts and rules are evaluated.
6. A decision (e.g., turn on heater) is made:
    - The action is published to an MQTT topic.
    - The new state is updated in the RDF model.
    - The RDF model is saved to the triple store.

6.8 Conclusion

This IoT thermostat system demonstrates a complete pipeline from secure communication and semantic modeling to real-time decision-making and device control. By combining MQTT, WoT, Drools, and Apache Jena, it achieves:

- Dynamic, rule-based behavior.
- Interoperability with other IoT devices through standardized descriptions.
- Semantic data persistence via RDF triple stores.
- Real-time feedback via MQTT topics.

Such a system is modular, extensible, and suitable for smart homes or building automation environments.

7 Overview of Temperature Simulator using MQTT and Python

This section explains a Python-based simulator that emulates a smart thermostat system using MQTT for communication. The simulator generates temperature values, reacts to heater status messages, and publishes temperature readings at regular intervals.

7.1 Logging and MQTT Configuration

Logging is set up to provide timestamps and log levels (e.g., INFO, ERROR). MQTT parameters such as the broker URL, port, username, password, topics, and client ID are defined to establish a secure connection with HiveMQ Cloud using TLS over port 8883.

7.2 Temperature Simulator Class

The core logic is encapsulated within a class that maintains:

- The current temperature.
- The heater state (ON/OFF).
- A direction variable to introduce momentum in temperature change.

Temperature is updated based on:

- Whether the heater is ON (temperature rises) or OFF (temperature falls).
- A base drift value with random variations.
- A momentum factor that probabilistically continues the last trend.

7.3 MQTT Callback Handlers

Three callback functions are defined:

- **on_connect**: Confirms successful connection and subscribes to relevant topics.
- **on_disconnect**: Logs unexpected disconnections and initiates automatic reconnection.
- **on_message**: Listens for heater status updates on the heater status topic and updates the simulator state.

7.4 MQTT Client Creation

An MQTT client instance is configured with the necessary credentials and callback bindings. TLS is enabled for secure communication, and reconnect delays are configured for robustness.

7.5 Simulation Loop

The main loop:

1. Connects to the MQTT broker and starts the network loop.
2. Repeatedly calculates a new temperature value using the simulator class.
3. Packages the temperature along with a timestamp and unit into a JSON payload.
4. Publishes this payload to the temperature topic using QoS level 1 to ensure at-least-once delivery.
5. Waits for a fixed interval (10 seconds) before the next iteration.

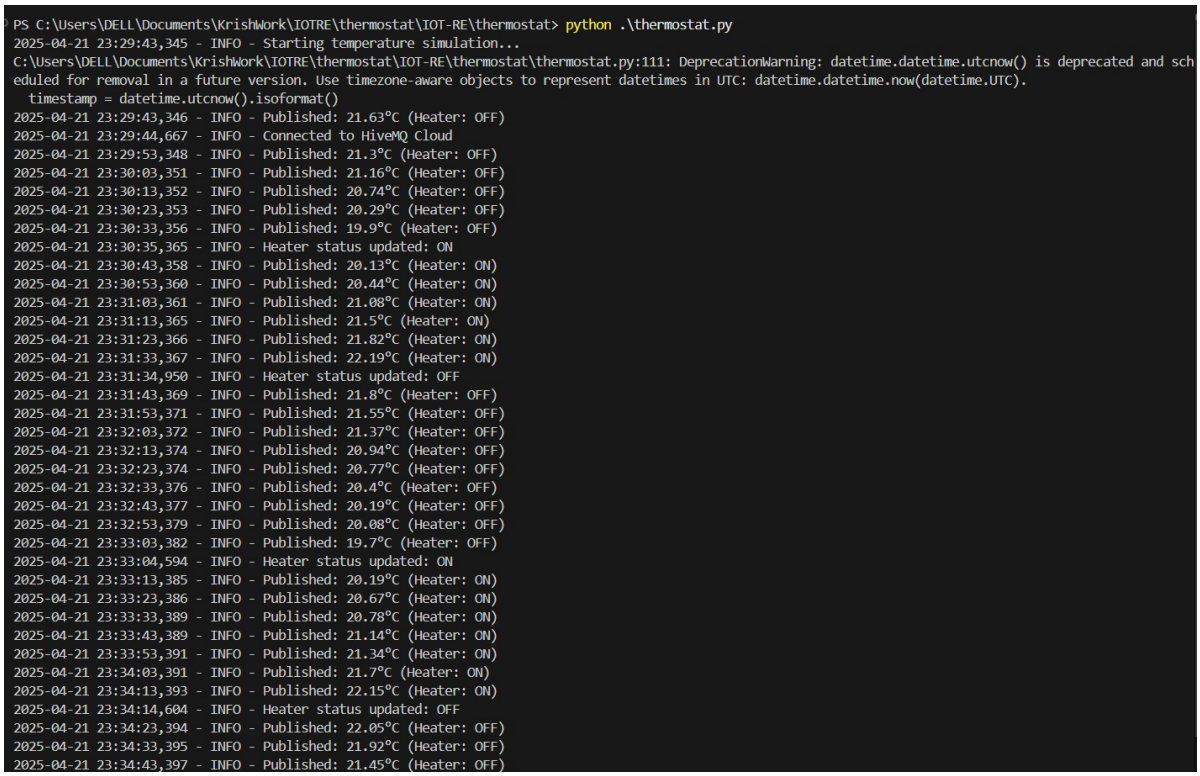

Figure 3: Python server publishing a new temperature every 10s and receiving heater status

7.6 Termination and Cleanup

The simulation gracefully handles keyboard interrupts and unexpected errors by stopping the network loop and disconnecting from the broker.

7.7 Conclusion

This implementation showcases a dynamic environment where temperature values adapt in real-time based on actuator input (heater ON/OFF), supporting interactive and realistic IoT behavior using MQTT and Python.

8 Thermostat Control using Drools Rules

The thermostat behavior is governed using two business rules written in Drools Rule Language (DRL). These rules monitor the current temperature and change the heater state accordingly.

8.1 Drools Rule Definitions

```
package rules

import com.iot.Temperature
import com.iot.ThermostatState

rule "Turn on heater when cold"
    when
        $temp : Temperature(value < 18)
        $state : ThermostatState(heaterOn == false)
    then
        System.out.println("Temperature is cold (" + $temp.getValue
    () + "°C), turning heater ON");
        $state.setHeaterOn(true);
end

rule "Turn off heater when warm enough"
    when
        $temp : Temperature(value > 22)
        $state : ThermostatState(heaterOn == true)
    then
        System.out.println("Temperature is warm (" + $temp.getValue
    () + "°C), turning heater OFF");
        $state.setHeaterOn(false);
end
```

8.2 Explanation of Rules

8.2.1 Rule 1: Turn on heater when cold

- **Condition (LHS):**
  - Temperature value is less than 18°C.
  - Heater is currently OFF.
- **Action (RHS):**
  - A log message is printed to indicate the low temperature.
  - The heater is turned ON by setting `heaterOn = true`.

8.2.2 Rule 2: Turn off heater when warm enough

- **Condition (LHS):**
  - Temperature value is greater than 22°C.
  - Heater is currently ON.
- **Action (RHS):**
  - A log message is printed to indicate the warm temperature.
  - The heater is turned OFF by setting `heaterOn = false`.

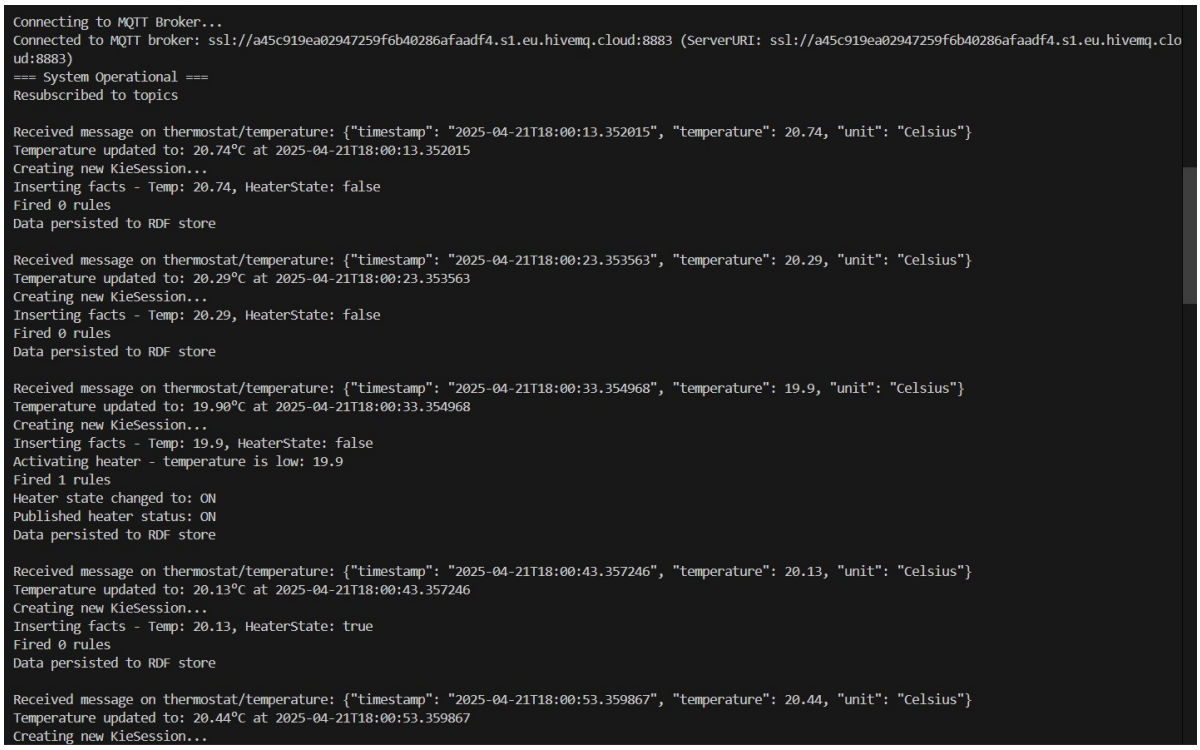

Figure 4: Output of Rules being applied

8.3 Working Principle

These rules define a simple reactive logic to manage heating:

- If the temperature drops below a comfortable threshold (18°C), the system turns the heater ON.
- If the temperature rises above a warm threshold (22°C), the system turns the heater OFF.

8.4 Integration with Java Classes

The rules rely on two Java classes:

- `Temperature`: Holds the current temperature value.
- `ThermostatState`: Maintains the current state of the heater (ON/OFF).

This rule-based system allows the thermostat to autonomously manage heating using declarative logic, improving clarity and maintainability.

9 RDF Representation of Temperature Reading

The following RDF snippet describes a temperature reading event from a thermostat device in Turtle syntax. It uses custom ontology terms under the namespace `http://iot.org/thermostat/`.

Figure 5: Sample Triplet stored in RDF format

9.1 Explanation of the RDF Format

This RDF triple set semantically represents a single temperature reading, structured as follows:

- **Subject:** `<http://iot.org/thermostat/Reading/1744137385292>` — A unique identifier (URI) for the temperature reading instance, typically derived from the timestamp.
- **Type:** The reading is classified as a `TemperatureReading`, meaning it is an instance of that RDF class.
- **heaterState:** `true` — Indicates that the heater was ON when the temperature was recorded.
- **timestamp:** `"1744137385292"^^xsd:long` — Represents the time of reading in UNIX epoch format with datatype as a long integer.
- **unit:** `"Â°C"` — Denotes the unit of the reading, intended to be degrees Celsius. Note: The character `Â` might be an encoding error; it should ideally be just `"°C"`.
- **value:** `"19.06"^^xsd:double` — Specifies the actual temperature value as a double-precision number.

10 Conclusion

This system demonstrates:

- Secure IoT device communication patterns
- Rule-based decision making in edge computing
- Semantic web integration for IoT
- Robust error handling and recovery

Future improvements could add OTA updates and energy consumption monitoring.