IoT Thermostat System Technical Documentation

Mentor: R Badrinath

Krish Dave (**IMT2022043**)

Vaibhav Mittal (**IMT2022126**)

Contents

1 Abstract **2**

2 System Features **2**
2.1 Core Features 2
2.2 Advanced Features 2

3 System Architecture **2**

4 Workflow **3**

5 Java Client Overview **3**
5.1 System Configuration 4
5.2 Main Method 4
5.3 Drools Initialization 4
5.4 Thing Description Definition 4
5.5 MQTT Setup 5
5.6 Message Handling 5
5.7 Updating RDF Triple Store 6
5.8 Heater Control and Publishing 6

6 Overview of Python server **6**
6.1 Logging Configuration 6
6.2 MQTT Configuration 6
6.3 Simulation Settings 6
6.4 MQTT Callback Functions 7
6.5 Temperature Simulation 7
6.6 MQTT Client Setup 7
6.7 Publishing Loop 7
6.8 Summary 8

7 Data Flow **8**

8 Thermostat Control using Drools Rules **8**
8.1 Drools Rule Definitions 9
8.2 Explanation of Rules 9
8.2.1 Rule 1: Turn on heater when cold 9
8.2.2 Rule 2: Turn off heater when warm enough 9
8.3 Working Principle 10
8.4 Integration with Java Classes 10

9 RDF Representation of Temperature Reading **10**
9.1 Explanation of the RDF Format 10

10 Conclusion **11**

1 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

2 System Features

2.1 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

2.2 Advanced Features

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

3 System Architecture

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

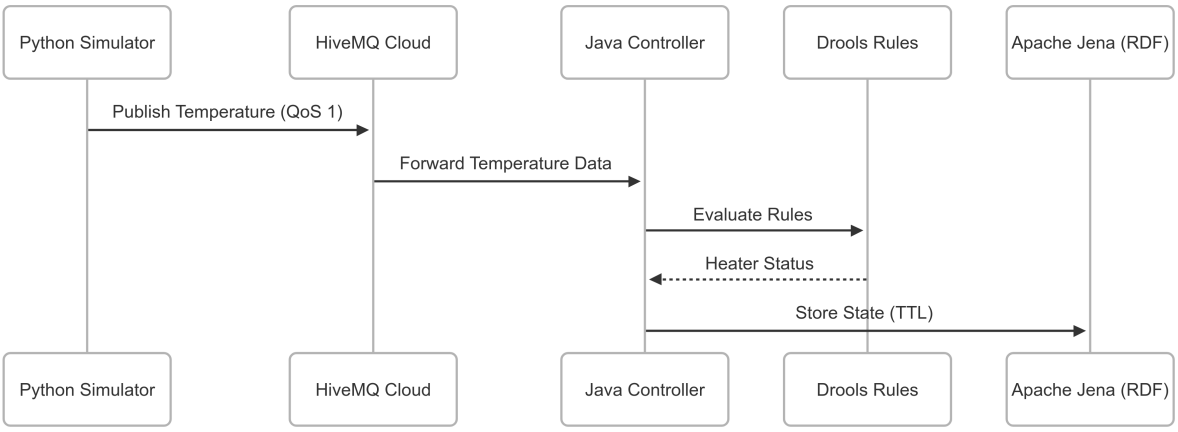

Figure 1: System Architecture Diagram

4 Workflow

1. Simulator generates temperature values (18-25°C ±0.5°C)
2. Python client publishes to `thermostat/temperature` via MQTT QoS 1
3. Java controller subscribes to temperature updates
4. System validates and processes messages using:
   - JSON schema validation
   - Timestamp verification
5. Drools engine evaluates rules against current state
6. Heater state updated based on rule evaluation
7. System persists state to RDF triple store

5 Java Client Overview

This Java program implements an enhanced IoT-based thermostat system integrating the following technologies:

- **MQTT** for real-time communication between devices.
- **Apache Jena** for storing and managing semantic RDF data.
- **WoT-TD (Web of Things - Thing Description)** to describe the capabilities and metadata of the thermostat.
- **Drools** rule engine for applying business logic based on incoming sensor data.

5.1 System Configuration

Global variables are defined to maintain the state of the system, such as the current temperature and whether the heater is on. MQTT topic strings are also defined here to facilitate structured communication between the device and its network.

5.2 Main Method

The `main` method serves as the entry point of the program. It initializes:

- The Drools rule engine,
- The Thing Description (TD) definition,
- The MQTT setup and connection.

5.3 Drools Initialization

This section sets up the Drools engine by loading rule files from the classpath. It creates a container and session, which are later used to evaluate the current state and apply rules such as whether to turn the heater on or off.

5.4 Thing Description Definition

The Thing Description is defined using the WoT-TD standard in JSON-LD format. It describes:

- The device's ID and context,
- Its properties (e.g., current temperature, heater status),
- Its actions (e.g., turn heater on/off),
- Events that the device may emit.

This TD can be used by other services or devices to understand the capabilities and structure of the thermostat.

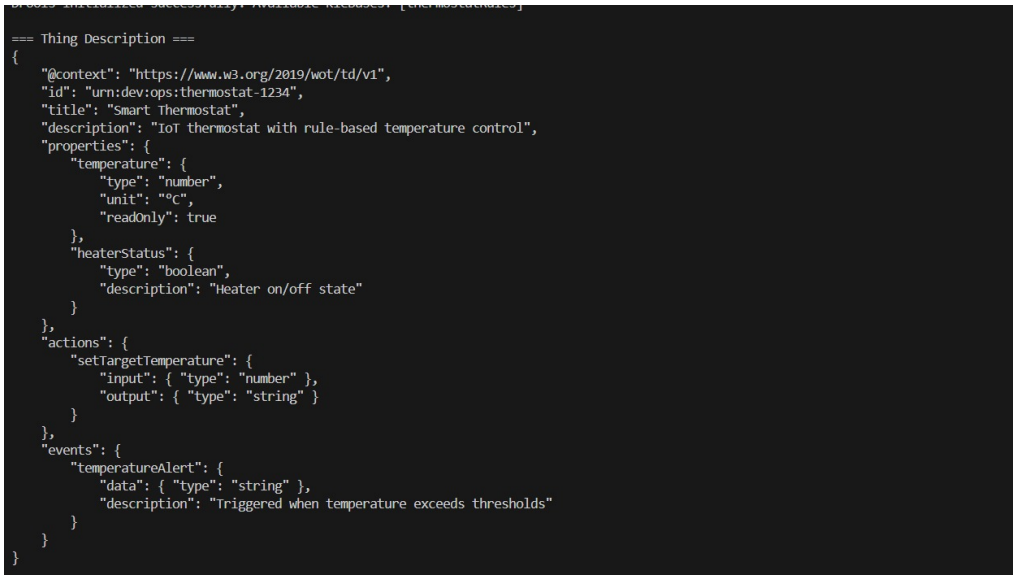

Figure 2: Sample Thing Description created

5.5 MQTT Setup

An MQTT client is created and configured using TLS for secure communication. It:

- Connects to the broker with credentials and client ID,
- Subscribes to a specific topic to receive temperature data,
- Defines callback functions for incoming messages and connection events.

5.6 Message Handling

When a message is received from the MQTT broker:

- It is parsed from JSON format to extract the temperature.
- The current state of the system is updated.
- An RDF triple representing the new state is created and stored using Apache Jena.
- The Drools engine is invoked with the updated facts to determine actions (e.g., activating the heater).

5.7 Updating RDF Triple Store

The system uses Apache Jena to persist temperature readings and heater states as RDF triples. This semantic data model allows other systems to query or reason about the thermostat's behavior using SPARQL or inference rules.

5.8 Heater Control and Publishing

Depending on the rule engine's output, the heater may be toggled on or off. This updated status is then published back to the MQTT broker, notifying other devices or systems of the new state.

6 Overview of Python server

This document explains the functionality of a Python script designed to simulate a thermostat by publishing temperature data every 10 seconds to a HiveMQ Cloud MQTT broker. The data is formatted in JSON and includes a timestamp, the temperature value, and the unit of measurement.

6.1 Logging Configuration

Logging is configured to include timestamps, log levels (INFO, ERROR, etc.), and messages. This helps monitor the script's behavior and debug issues.

6.2 MQTT Configuration

Several constants are defined for MQTT communication:

- `BROKER`: The HiveMQ Cloud broker address.
- `PORT`: MQTT over TLS uses port 8883.
- `USERNAME, PASSWORD`: Credentials for secure connection.
- `TOPIC`: The MQTT topic to publish data to.
- `CLIENT_ID`: A unique ID generated using random numbers to avoid session conflicts.

6.3 Simulation Settings

Defines the range for simulated temperatures:

- `MIN_TEMP = 18.0`, `MAX_TEMP = 25.0`
- `INTERVAL_SEC = 10`: The interval in seconds between each temperature reading.

6.4 MQTT Callback Functions

- `on_connect`: Logs connection status. If successful, subscribes to the topic.
- `on_disconnect`: Handles unexpected disconnections and attempts reconnection.

6.5 Temperature Simulation

`get_simulated_temperature` generates a random temperature in the defined range with a small fluctuation. The result is rounded to two decimal places for realism.

6.6 MQTT Client Setup

`create_mqtt_client` creates and configures the MQTT client:

- Sets the client ID, credentials, and TLS encryption.
- Binds callback functions.
- Enables automatic reconnects with exponential backoff.

6.7 Publishing Loop

`run_simulation` is the core loop of the program:

1. Connects to the broker and starts the MQTT loop in a separate thread.
2. In a continuous loop:
    - Gets current UTC time in ISO format.
    - Generates a new simulated temperature.
    - Formats the data as a JSON payload.
    - Publishes to the topic with QoS 1 for at-least-once delivery.
    - Logs success/failure.
    - Waits for the defined interval before repeating.
3. Handles KeyboardInterrupt for graceful shutdown.
4. Ensures proper disconnection and cleanup at the end.

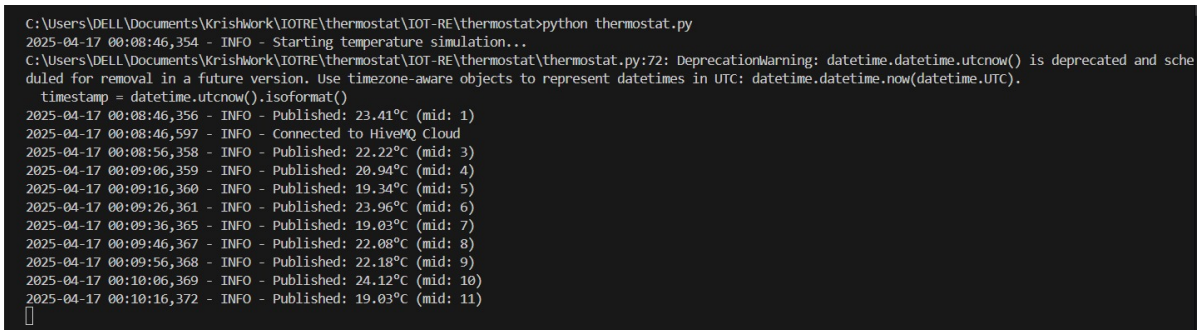

Figure 3: Python server publishing a new temperature every 10s

6.8 Summary

This Python script acts as a thermostat simulator. It publishes temperature readings every 10 seconds to a secure MQTT topic on HiveMQ Cloud. The readings are structured as JSON and timestamped for use in IoT applications or real-time monitoring systems. Features include secure TLS connection, automatic reconnects, logging, and realistic simulation of temperature values.

7 Data Flow

$$T_{current} \rightarrow \text{MQTT} \rightarrow \text{Drools} \rightarrow \begin{cases} \text{Heater State} \\ \text{RDF Storage} \\ \text{Status Updates} \end{cases} \tag{1}$$

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`.

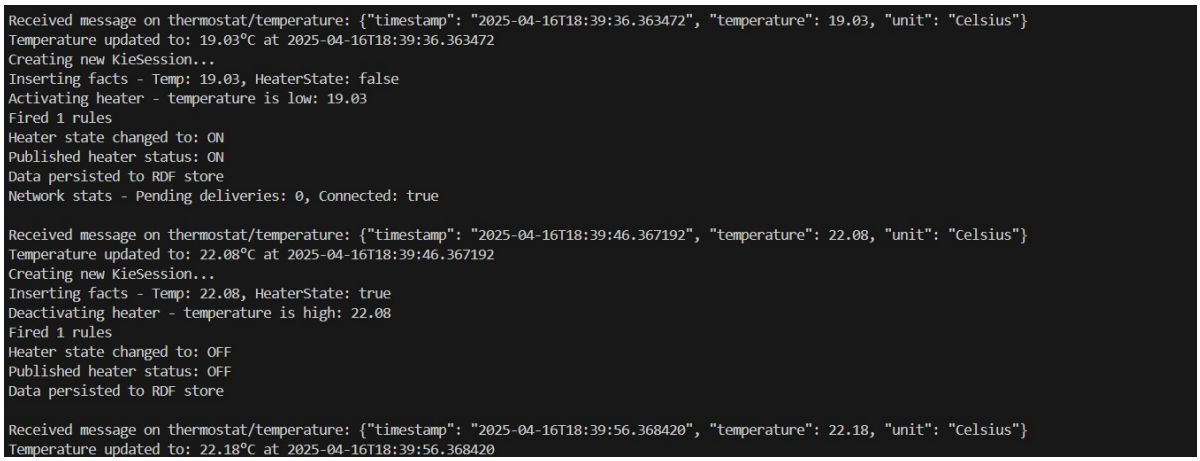

Figure 4: Output of Rules being applied

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/`.

9.1 Explanation of the RDF Format

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

Figure 5: Sample Triplet stored in RDF format

- **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.