Smart Climate Control System – CA Report

Student ID: 23413271

Student Name: Guilherme Junio da Silveira

Course: HDCSDEV_INT

Module: Distributed Systems

Lecturer: Catriona Nic Lughadha

Institution: National College of Ireland – NCI

Table of Contents

Domain Description

The assigned domain for this project is **Smart Home Automation**, a system that aims to improve home comfort, safety, and energy efficiency. The Smart Climate Control System simulates a smart home environment built using Java and gRPC to demonstrate core distributed systems concepts. Its purpose is to show how real-world smart devices are represented through gRPC-based services and interact with clients through a central control interface. The project implements all four types of gRPC communication styles and presents secure, modular, and interactive simulation of device control via graphical user interfaces (GUI's).

Description of the Three Core Services

The Smart Climate Control System presents three interactive gRPC services. Each service runs on a separate server using its own gRPC port. Together, they form a distributed system architecture that demonstrates modular design and real-time interaction.

Thermostat Service

Function: Sets and retrieves the current temperature, enables/disables auto-adjust mode, and streams periodic temperature updates to the client.

Contribution: Acts as the main controller for heating and cooling simulation. It gives users the ability to monitor and change temperature settings manually or through automation.

RPC Styles: Unary (set/get temperature, auto-adjust toggle), Server Streaming (periodic temperature updates).

Humidity Control Service

Function: Accepts a stream of humidity readings from the client and responds with an aggregated status message presenting the average humidity.

Contribution: Helps simulate how smart humidifiers or dehumidifiers might assess and react to ongoing changes in humidity over time.

RPC Style: Client Streaming (send multiple humidity readings, receives single summary response).

Air Quality Monitor Service

Function: Allows clients to send room names in a steam and receive multiple air quality alerts asynchronously for each room.

Contribution: Provides real-time air quality monitoring across various rooms in a house, helping simulate alerts for smoke, CO2 levels, or ventilation suggestions.

RPC Style: Bi-Directional Streaming (stream requests and receives alerts simultaneously).

Service Definitions and RPC

This section outlines the detailed definitions of each gRPC service implemented in the system. It describes the request and response message structures, RPC methods, and types of communication pattern used in each case.

Thermostat Service

Service Name: `Thermostat`

RPC Methods:

1. **SetTemperature (Unary):** Allows the client to set a temperature manually
   - Request: `TemperatureRequest {float temperature}`
   - Response: `TemperatureResponse {float currentTemperature}`
2. **GetCurrentTemperature (Unary):** Fetches current temperature maintained by the server
   - Request: `Empty {}`
   - Response: `TemperatureResponse {float currentTemperature}`
3. **StreamTemperatureUpdates (Server Streaming):** Continuously streams simulated temperature updates to the client over time
   - Request: `Empty {}`
   - Response: `stream TemperatureResponse {float currentTemperature}`
4. **AutoAdjustMode (Unary):** Enables or disables auto-adjust mode for temperature control
   - Request: `AutoAdjustRequest {bool enable}`
   - Response: `StatusResponse {string message}`

Humidity Control Service

Service Name: `HumidityControl`

RPC Method:

1. **SetHumidityLevel (Client Streaming):** Accepts multiple humidity readings and returns a message summarizing the number of values received and their average.

- Request: `stream HumidityRequest {float humidity}`
- Response: `statusResponse {string message}`

Air Quality Monitor Service

Service Name: `AirQualityMonitor`

RPC Methods:

1. **MonitorAirQuality (Bi-Directional Streaming):** For each room name received, the server responds with a sequence of air quality alerts. This simulates real-time environmental monitoring and alerts.
    - Request: `stream AirQualityCheck {string location}`
    - Response: `stream AirQualityAlert {string alertMessage}`

Each service is designed to cover one of the four gRPC communication styles:

- **Unary:** Simple request/response (Thermostat)
- **Server Streaming:** Continuous server push (Thermostat)
- **Client Streaming:** Batch client push with single response (Humidity Control)
- **Bi-Directional Streaming:** Continuous client and server push, real-time communication (Air Quality Monitor)

Service Implementations

Each service in the system is implemented as a dedicated Java class extending its corresponding gRPC base class. These classes define the actual behaviour and logic.

ThermostatServiceImpl.java

Implements the logic for all Thermostat RPCs:

- `setTemperature`: Saves a new temperature and responds with the updated value.
- `getCurrentTemperature`: Returns the current stored temperature.
- `streamTemperatureUpdates`: Sends periodic updates (Simulated every second).
- `autoAdjustMode`: Toggles automatic mode and returns a confirmation message.

Security is handled using JWT-based server-side interceptors to validate client tokens.

HumidityServiceImpl.java

Implements a Client-Streaming method:

- `setHumidityLevel`: Collects multiple humidity readings, calculates the average, and responds with a summary containing the number of readings and average humidity.

A simple list is used to buffer values before generating the final response.

AirQualityServiceImpl.java

Implements bi-directional communication:

- `monitorAirQuality`: For each room sent by the client, the server returns multiple air quality alerts using delayed responses to simulate real-time changes.

A helper method (`delayedAlert`) spawns background threads to stream simulated alerts after delays, enhancing realism.

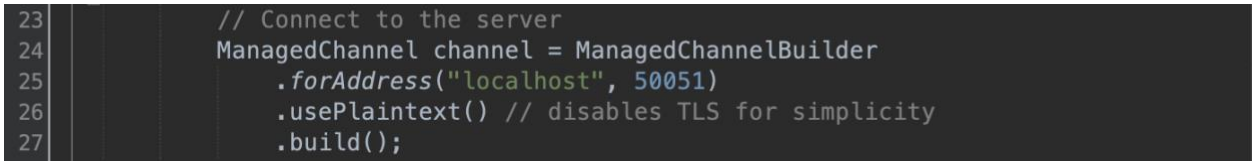
Each service is independently hosted on its own server instance and port, enabling full modularity. Servers register interceptors to enforce JWT-based authentication.

Use of Naming Services

As mentioned before, each of the three services is hosted on a different port and independently launched using its own server. To simulate naming and service discovery in a simplified local environment, the client applications (GUIs) connect directly to services using hardcoded localhost addresses and specific port numbers:

- Thermostat Service: `localhost:50051`
- Humidity Control Service: `localhost:50052`
- Air Quality Monitor Service: `localhost:50053`

This static setup emulates the concept of naming services in distributed systems, where clients need a way to locate and connect to remote services. Each GUI or client component knows which service it needs to talk to and uses the corresponding port, demonstrating a basic form of service location strategy. See the example below:



```
        // Connect to the server
        ManagedChannel channel = ManagedChannelBuilder
            .forAddress("localhost", 50051)
            .usePlaintext() // disables TLS for simplicity
            .build();
```

Figure 1: Connecting to the server using port 50051 in ThermostatClient.java file.

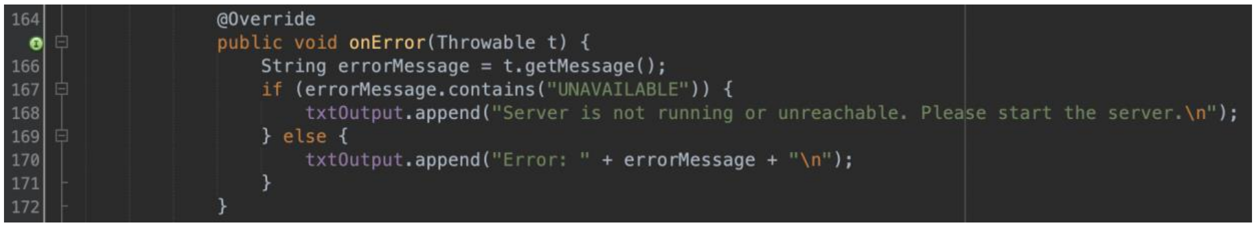
Error Handling and Advanced Features

Error Handling in gRPC Services

All three services implement robust error handling strategies to ensure system stability and provide meaningful feedback to users. The following practices were adopted:

GUI-Level User-Friendly Feedback

Each GUI displays human-readable error messages in its text area, helping the user understand what is going wrong, especially when services are unavailable. See the example below:



```
@Override
public void onError(Throwable t) {
    String errorMessage = t.getMessage();
    if (errorMessage.contains("UNAVAILABLE")) {
        txtOutput.append("Server is not running or unreachable. Please start the server.\n");
    } else {
        txtOutput.append("Error: " + errorMessage + "\n");
    }
}
```

Figure 2: Error handling in HumidityGUI – Server Not Running

This logic avoids exposing low-level technical exceptions and instead guides users to take the correct actions.

Stream Termination and Exception Safety

Each streaming method gracefully handles termination:

- `responseObserver.onCompleted()` is called when appropriate.
- Any `InterruptedException` or other runtime errors and caught and passed to `onError().`

This prevents the system from hanging or leaking resources during bi-directional or client/server streaming sessions.

Client GUI

To provide user-friendly access to the gRPC services, each component of the system includes a GUI, built using NetBeans GUI Builder (Matisse) and the Swing framework/library. These graphical interfaces simulate real-world smart home dashboards, allowing users to interact with the backend services via buttons, dropdowns, and text fields. Each GUI uses a different type of gRPC communication style.

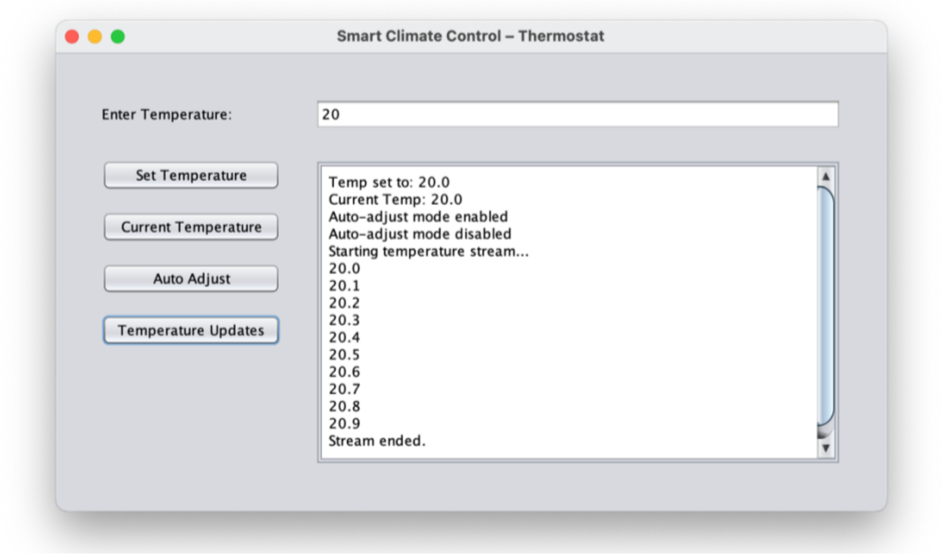
Thermostat GUI

Communication Style: Unary and Server Streaming

Features:

- Set and get the current temperature
- Toggle auto-adjust mode
- Stream temperature updates

Screenshot:



```
Temp set to: 20.0
Current Temp: 20.0
Auto-adjust mode enabled
Auto-adjust mode disabled
Starting temperature stream...
20.0
20.1
20.2
20.3
20.4
20.5
20.6
20.7
20.8
20.9
Stream ended.
```

Figure 3: Thermost GUI

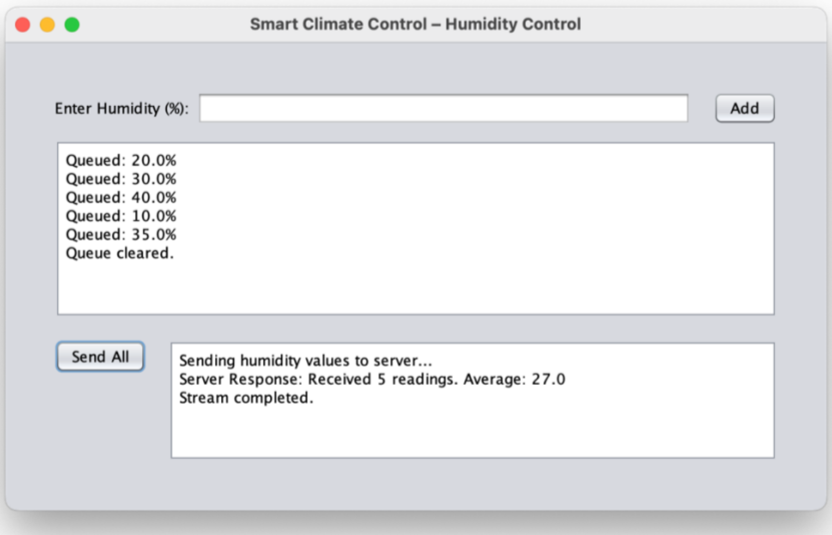
Humidity Control GUI

Communication Style: Client Streaming

Features:

- Queue multiple humidity values from user input
- Send all values to the server
- Receive average humidity and feedback

Screenshot:



```
Queued: 20.0%
Queued: 30.0%
Queued: 40.0%
Queued: 10.0%
Queued: 35.0%
Queue cleared.
```

```
Sending humidity values to server...
Server Response: Received 5 readings. Average: 27.0
Stream completed.
```

Figure 4: Humidity Control GUI

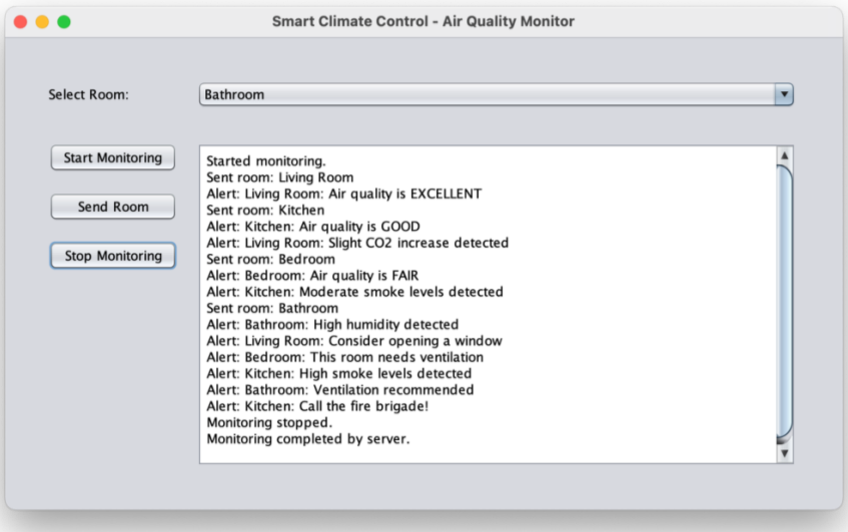
Air Quality Monitor GUI

Communication Style: Bi-Directional Streaming

Features:

- Start monitoring session
- Select room from dropdown and send it
- Receive real-time alerts for air quality changes in each room

Screenshot:



```
Started monitoring.
Sent room: Living Room
Alert: Living Room: Air quality is EXCELLENT
Sent room: Kitchen
Alert: Kitchen: Air quality is GOOD
Alert: Living Room: Slight CO2 increase detected
Sent room: Bedroom
Alert: Bedroom: Air quality is FAIR
Alert: Kitchen: Moderate smoke levels detected
Sent room: Bathroom
Alert: Bathroom: High humidity detected
Alert: Living Room: Consider opening a window
Alert: Bedroom: This room needs ventilation
Alert: Kitchen: High smoke levels detected
Alert: Bathroom: Ventilation recommended
Alert: Kitchen: Call the fire brigade!
Monitoring stopped.
Monitoring completed by server.
```

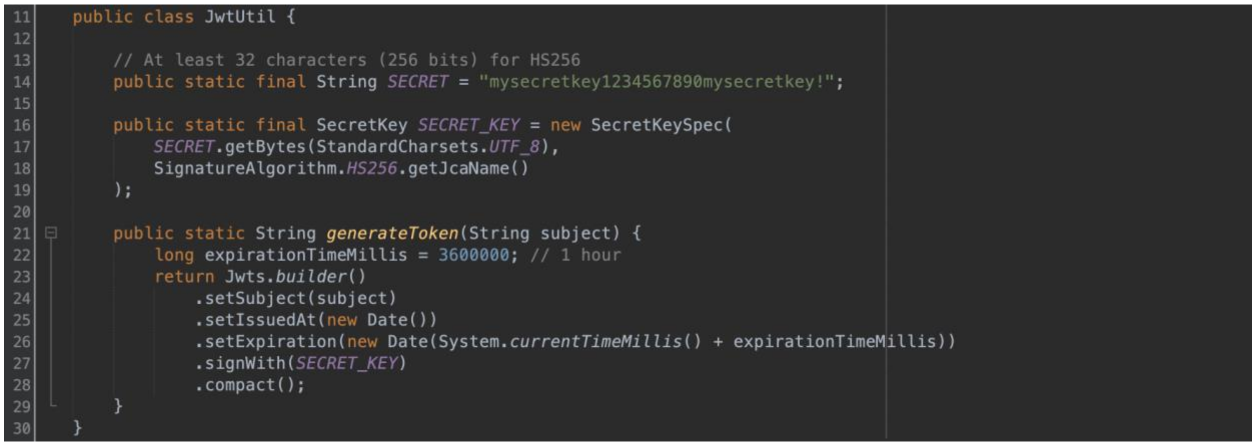
Figure 5: Air Quality Monitor GUI

Security Features

To secure communication between clients and servers, a JWT-based authentication mechanism is implemented using a shared secret key. See the examples:

1. Token Generation (JwtUtil.java)

- Generates a signed JWT using HS256 and 256-bit secret key.
- Used on the client side before sending any request.

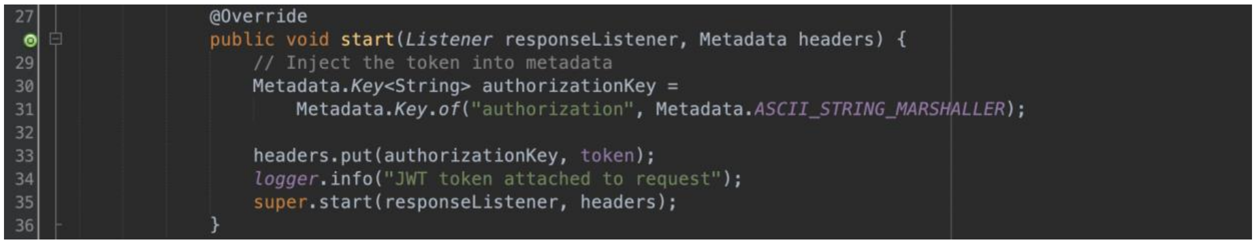


```java
public class JwtUtil {

    // At least 32 characters (256 bits) for HS256
    public static final String SECRET = "mysecretkey1234567890mysecretkey!";

    public static final SecretKey SECRET_KEY = new SecretKeySpec(
        SECRET.getBytes(StandardCharsets.UTF_8),
        SignatureAlgorithm.HS256.getJcaName()
    );

    public static String generateToken(String subject) {
        long expirationTimeMillis = 3600000; // 1 hour
        return Jwts.builder()
            .setSubject(subject)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + expirationTimeMillis))
            .signWith(SECRET_KEY)
            .compact();
    }
}
```

Figure 6: Token generation in JwtUtil.java

2. Attach Token to Request (JwtClientInterceptor.java)

- Adds the JWT to gRPC request metadata.
- Automatically included in every client call.

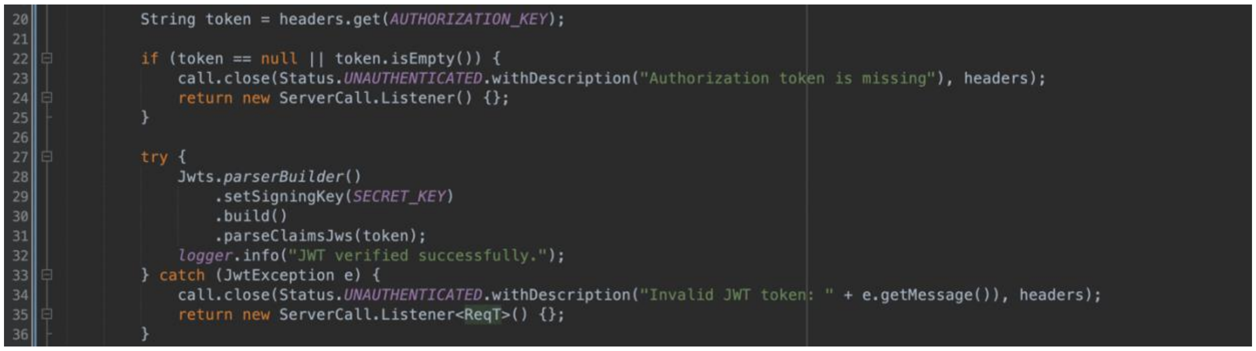


```java
            @Override
            public void start(Listener responseListener, Metadata headers) {
                // Inject the token into metadata
                Metadata.Key<String> authorizationKey =
                    Metadata.Key.of("authorization", Metadata.ASCII_STRING_MARSHALLER);

                headers.put(authorizationKey, token);
                logger.info("JWT token attached to request");
                super.start(responseListener, headers);
            }
```

Figure 7: Attaching token to request in JwtClientInterceptor.java

3. Verify Token (JwtServerInterceptor.java)

- Extracts the JWT from the incoming request.
- Rejects unauthenticated calls with a proper error it the token is invalid or missing.



```java
        String token = headers.get(AUTHORIZATION_KEY);

        if (token == null || token.isEmpty()) {
            call.close(Status.UNAUTHENTICATED.withDescription("Authorization token is missing"), headers);
            return new ServerCall.Listener() {};
        }

        try {
            Jwts.parserBuilder()
                .setSigningKey(SECRET_KEY)
                .build()
                .parseClaimsJws(token);
            logger.info("JWT verified successfully.");
        } catch (JwtException e) {
            call.close(Status.UNAUTHENTICATED.withDescription("Invalid JWT token: " + e.getMessage()), headers);
            return new ServerCall.Listener<ReqT>() {};
        }
```

Figure 8: Verifying token in JwtServerInterceptor.java

GitHub Repository (Version Control)

This project is managed using Git and hosted on GitHub, ensuring proper version control and traceability. Each stage of development was tracked through meaningful commits.

Repository link: GitHub - Smart Climate Control System

Conclusion

The Smart Climate Control System successfully demonstrates key principles of distributed systems through the implementation of three modular, gRPC-based services. Each service operates independently, showcases a unique gRPC communication style, and interacts with clients through a user-friendly GUI.

Throughout the development of this project, concepts such as service modularity, RPC definitions, server/client architecture, JWT-based security, and streaming mechanisms were explored and applied in practice. The use of JWT added a layer of secure communication, ensuring that only authenticated clients can access the services.

This experience provided valuable insights into gRPC frameworks, secure service communication, error handling, and the fundamentals of distributed software design. It also helped strengthen practical Java development skills and introduced best practices in modular programming and version control using GitHub.