Final Project Report

IoTThings ServiceIDE – High-Level Functionality Report

1. Overview

IoTThings ServiceIDE is a graphical Integrated Development Environment (IDE) for designing, managing, and executing Internet of Things (IoT) applications within the paradigm of smart spaces. It enables automatic discovery of devices (referred to as *Things*), their components and services, and provides tools for defining inter-service relationships and composing applications through a visual interface.

2. System Architecture and Workflow

2.1. Startup and Discovery

Upon launch, the application initiates a background thread called `TweetListener`, which listens for multicast JSON messages, referred to as *tweets*, broadcast by IoT devices on the local network. These messages provide metadata about:

- **Things**: Devices present in the network.
- **Entities**: Logical components within a device.
- **Services**: Functionalities offered by entities.
- **Relationships**: Execution logic connecting services.

Each incoming tweet is parsed and used to update the internal `IoTContext`, a centralized structure that maintains the current network state and all discovered components.

2.2. Graphical User Interface (GUI)

The GUI is structured into four main tabs:

1. **Things**: Displays all discovered devices with their properties and contained entities.
2. **Services**: Lists all known services with detailed information.
3. **Relationships**: Shows logical dependencies between services.
4. **Apps**: Facilitates the creation, editing, and execution of IoT applications.

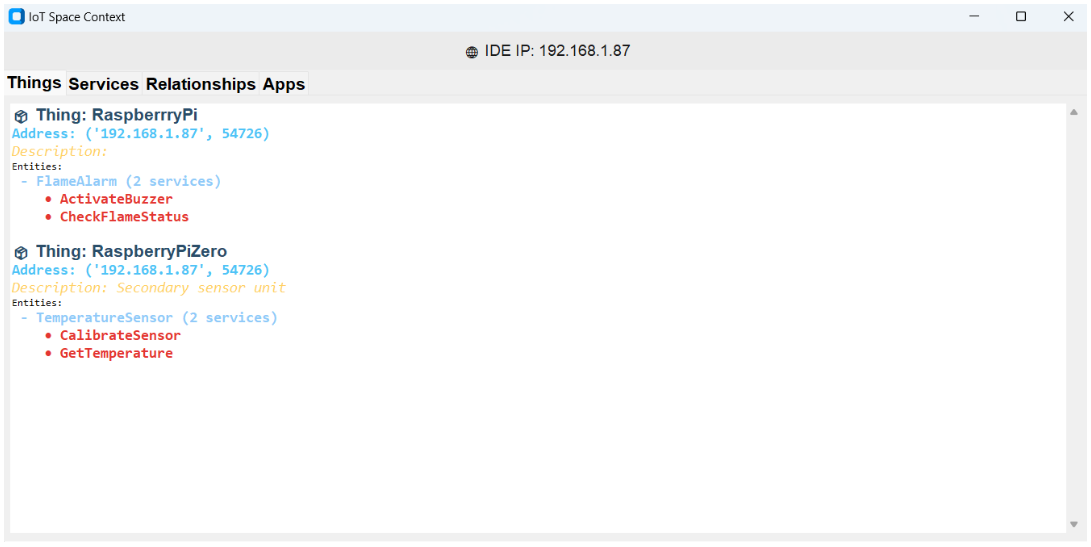
2.3. Thing, Entity, and Service Management

Things represent networked IoT devices. Each Thing comprises one or more **Entities**, which can be sensors, actuators, or logical modules. Each Entity may expose one or more **Services**. Services have associated metadata, including API type, input/output parameters, and documentation.

2.4. Service Relationships

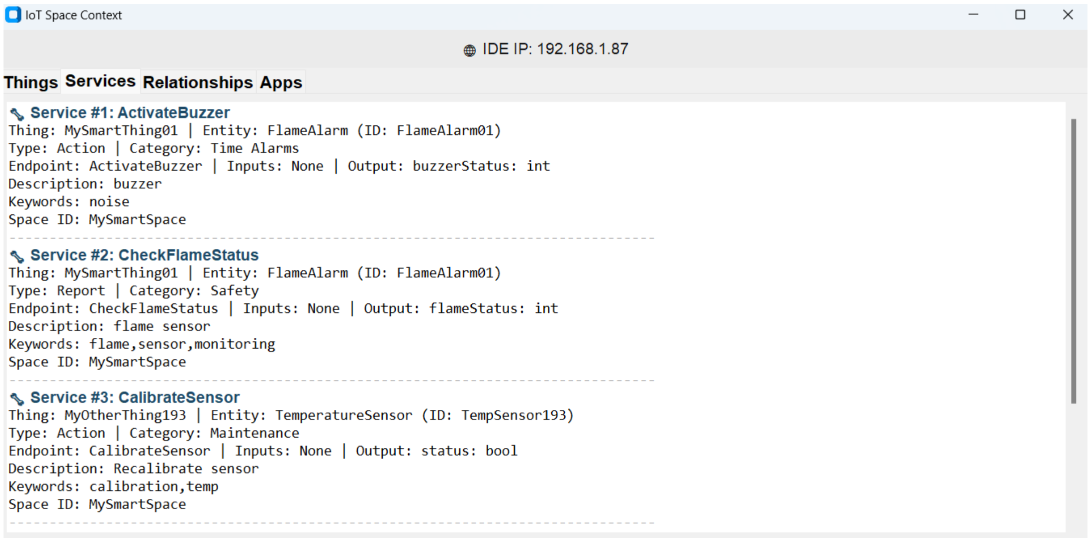
Services can be interconnected using logical relationships, defined in the **Relationships** tab*. Supported types include:

- **Ordered**: Executes the destination service after the source.
- **On-success**: Executes the destination only if the source completes successfully.
- **Conditional**: Executes the destination only if a specific condition on the source's output holds.



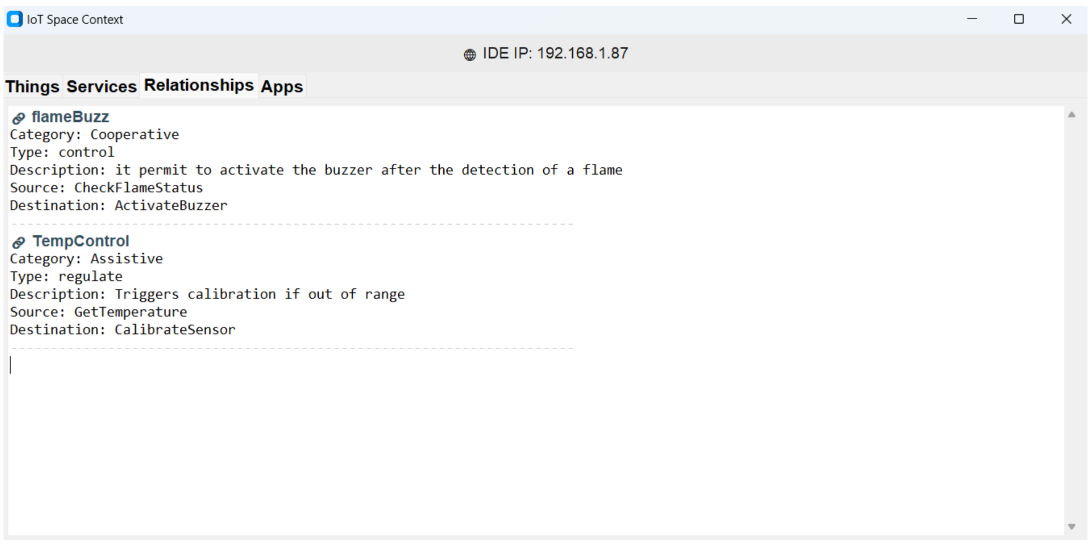
```
Thing: RaspberrryPi
Address: ('192.168.1.87', 54726)
Description:
Entities:
 - FlameAlarm (2 services)
    • ActivateBuzzer
    • CheckFlameStatus

Thing: RaspberryPiZero
Address: ('192.168.1.87', 54726)
Description: Secondary sensor unit
Entities:
 - TemperatureSensor (2 services)
    • CalibrateSensor
    • GetTemperature
```

Figure 1: Things tab: overview of discovered devices and entities



```
Service #1: ActivateBuzzer
Thing: MySmartThing01 | Entity: FlameAlarm (ID: FlameAlarm01)
Type: Action | Category: Time Alarms
Endpoint: ActivateBuzzer | Inputs: None | Output: buzzerStatus: int
Description: buzzer
Keywords: noise
Space ID: MySmartSpace
--------------------------------------------------------------------------------
Service #2: CheckFlameStatus
Thing: MySmartThing01 | Entity: FlameAlarm (ID: FlameAlarm01)
Type: Report | Category: Safety
Endpoint: CheckFlameStatus | Inputs: None | Output: flameStatus: int
Description: flame sensor
Keywords: flame,sensor,monitoring
Space ID: MySmartSpace
--------------------------------------------------------------------------------
Service #3: CalibrateSensor
Thing: MyOtherThing193 | Entity: TemperatureSensor (ID: TempSensor193)
Type: Action | Category: Maintenance
Endpoint: CalibrateSensor | Inputs: None | Output: status: bool
Description: Recalibrate sensor
Keywords: calibration,temp
Space ID: MySmartSpace
--------------------------------------------------------------------------------
```

Figure 2: Services tab: detailed view of available services



flameBuzz

```
Category: Cooperative
Type: control
Description: it permit to activate the buzzer after the detection of a flame
Source: CheckFlameStatus
Destination: ActivateBuzzer
```

TempControl

```
Category: Assistive
Type: regulate
Description: Triggers calibration if out of range
Source: GetTemperature
Destination: CalibrateSensor
```

Figure 3: Relationships tab: detailed view of the relationships

3. Application Composition and Execution

3.1. App Construction

Users can access the **Apps** tab to create or edit IoT applications using a dedicated **Graphical App Editor**. Functionalities include:

- Drag-and-drop of available services onto a canvas.
- Definition of relationships between selected services.
- Configuration of service parameters.
- Saving and loading of application projects.

The editor comprises:

- A list of available services.
- A canvas area for placing service nodes.
- An input panel for configuring parameters.
- A legend explaining relationship types.

3.2. App Execution

Applications can be executed directly from the IDE. The system respects all defined relationships and execution constraints. A terminal-like interface logs execution progress. Execution can be interrupted at any time.

Applications are persisted as `.iot` files (in JSON format) and may be reloaded or uploaded.

4. User Workflow

1. Startup

- Launch the IDE.
- Automatic discovery of Things, Entities, and Services.

2. Exploration

- Browse the **Things** and **Services** tabs to explore available resources.

3. Application Design

- Navigate to the **Apps** tab and select `Start New App`.
- Use the Graphical Editor to add services and define relationships (e.g., "If temperature ¿ 30°C, turn on fan").
- Configure service inputs.
- Save the app.

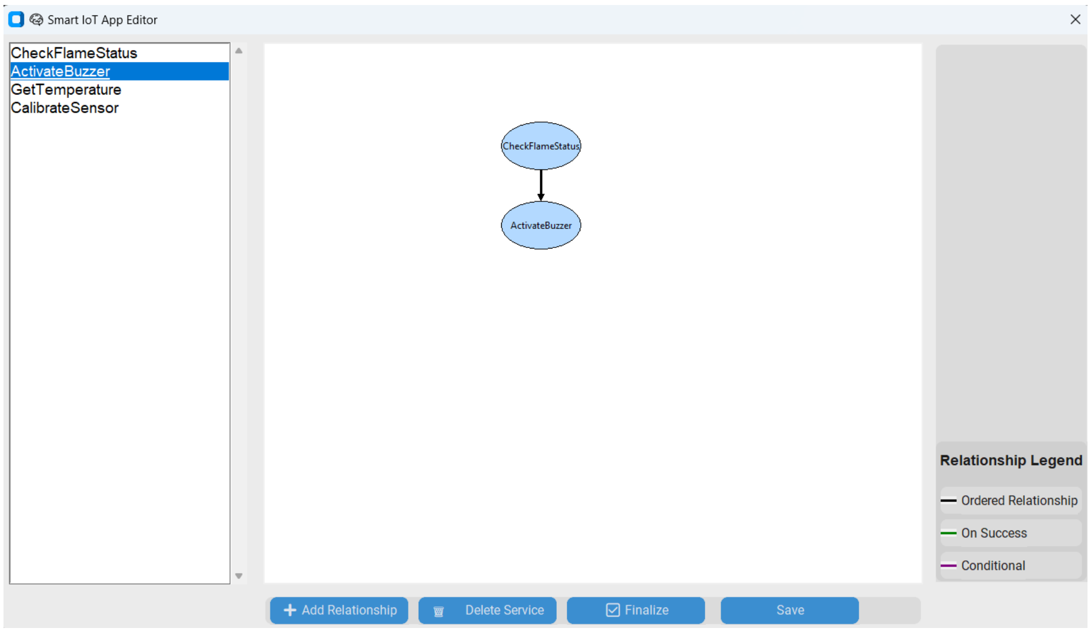


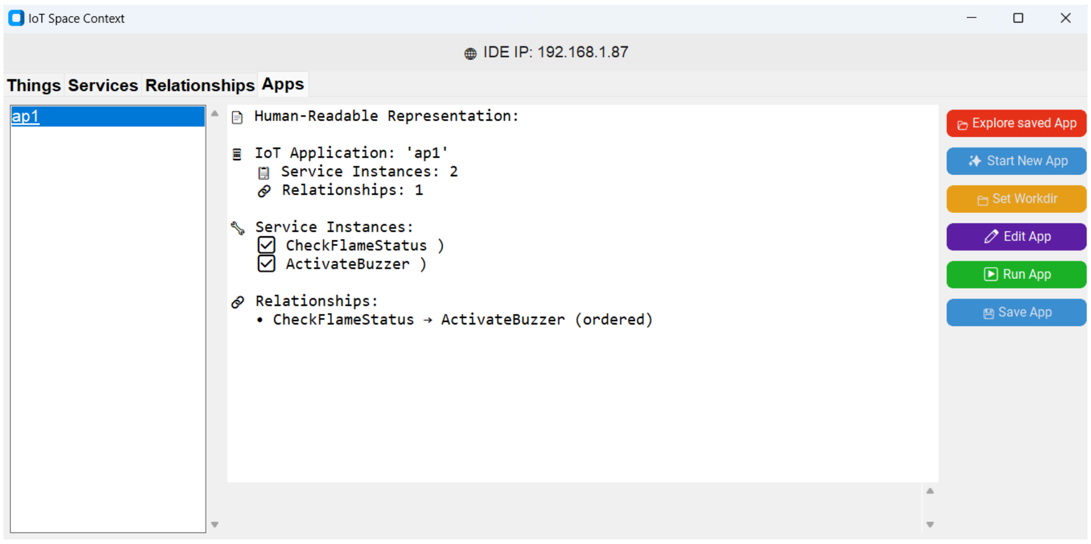
Figure 4: Graphical App Editor: drag-and-drop canvas and configuration panels

4. Execution and Monitoring

- Select an app and click `Run`.
- Follow monitor logs.
- Stop execution as needed.

5. Application Management

- Apps can be saved, reloaded, or deleted.



```
Human-Readable Representation:

IoT Application: 'ap1'
  Service Instances: 2
  Relationships: 1

Service Instances:
  ☑ CheckFlameStatus )
  ☑ ActivateBuzzer )

Relationships:
  • CheckFlameStatus → ActivateBuzzer (ordered)
```

Figure 5: Apps tab: project management and execution interface

5. Technical Notes

- **Extensibility**: Modular architecture allows future support for new devices and relationship types.
- **Persistence**: Projects are saved in portable JSON-based `.iot` files.
- **Networking**: Device discovery is based on UDP multicast communication.
- **Third-party software**: Third-party software was used only for the integration of a graphical interface.

6. Conclusion

IoTThings ServiceIDE offers a powerful, user-friendly platform for the graphical design and orchestration of IoT applications in dynamic smart spaces. By combining automatic discovery, drag-and-drop composition, and real-time execution feedback, the IDE empowers users to develop and manage complex workflows across heterogeneous IoT devices with minimal effort.

Relationships implementation problems*

The types of relationships supported and implemented (as required by the project) differ from those announced by Atlas through its periodic tweets. While it would have been possible to define a mapping between Atlas relationships and those implemented in the system, such a mapping would not have fully captured the semantics of all the relationships announced by the platform. Therefore, we chose not to implement any predefined service relationships, nor to impose constraints on the user's choice of relationships—even when they conflict with those announced by Atlas. This decision was made to prevent the application from becoming buggy or malfunctioning. Nevertheless, the integration of such mapping logic could be considered in future developments of the application.