Raspberry Pi Connector: Architecture and Functions Explanation

Overall Architecture

The Raspberry Pi Connector serves as a critical bridge in the Smart IoT Bolt system, connecting physical sensors and actuators with the broader microservices ecosystem. It fulfills multiple roles specified in the project documentation and implements a modular, fault-tolerant design.

```
+--------------------------------------------+
|              Raspberry Pi Connector        |
+--------------------------------------------+
|                                            |
|  +-------------+      +---------------+    |
|  | Sensor      |----->| MQTT Service  |---+---> MQTT Broker
|  | Simulation  |      |               |   |       |
|  +-------------+      +---------------+    |       |
|        ^                    ^              |       |
|        |                    |              |       |
|  +-------------+      +---------------+    |       v
|  | REST API    |<---->| Actuator      |<--+--- Control Center
|  | (CherryPy)  |      | Service       |    |
|  +-------------+      +---------------+    |
|        |                    ^              |
|        v                    |              |
|  +------------------------------------------+
|  |            Catalog Manager              |----> Resource Catalog
|  +------------------------------------------+
|                                            |
+--------------------------------------------+
```

Core Components

The connector is structured into several focused modules, each handling a specific aspect of functionality:

1. **Main Coordinator (`main.py`)**
   - Orchestrates startup and shutdown sequences
   - Manages lifecycle of all internal services
   - Implements fault-tolerance through retry mechanisms
2. **Sensor Simulation (`sensor_service.py`)**
   - Simulates temperature and pressure sensors

    - Generates readings using Gaussian distribution
    - Publishes data at configurable intervals
3. **Actuator Control (`actuator_service.py`)**
    - Manages valve state (open/closed)
    - Processes control commands
    - Tracks state changes with timestamps
4. **MQTT Communication (`mqtt_service.py`)**
    - Establishes and maintains broker connection
    - Publishes sensor data to appropriate topics
    - Subscribes to valve control commands
5. **REST API (`rest_api.py`)**
    - Exposes endpoints using CherryPy
    - Provides interfaces for sensor data retrieval
    - Enables valve control via REST
6. **Catalog Integration (`catalog_manager.py`)**
    - Handles service registration
    - Retrieves configuration parameters
    - Updates sensor and actuator status

Detailed Function Explanation

1. Data Publishing Function

According to the PDF, the Raspberry Pi Connector "acts as a data publisher, collecting temperature and pressure readings from sensors and monitoring pipeline conditions."

Implementation:

- The `SensorService` class generates simulated sensor data using Gaussian distribution (via `utils/gaussian.py`)
- The `_generate_sensor_readings()` method creates temperature and pressure values based on configurable mean and standard deviation
- The `_publish_sensor_data()` method runs in a dedicated thread, continuously generating and publishing readings at the configured interval
- Data is published to MQTT topics via the `MQTTService.publish_temperature()` and `MQTTService.publish_pressure()` methods

Code Process:

1. Sensor data is generated using statistical models to mimic real-world variability
2. A timestamp is attached to each reading
3. The data is formatted into JSON payloads
4. The MQTT service publishes the data to the respective topics
5. The same data is also sent to the Resource Catalog for system-wide awareness

2. Valve Connection Function

The PDF states that the connector "connects to the valve actuator (wired connection) to respond during emergencies."

Implementation:

- The `ActuatorService` class manages the valve state
- The `set_valve_state()` method changes the valve state when commanded
- The `get_valve_state()` method provides the current state and timestamp

Code Process:

1. The actuator service maintains the current valve state ("open" or "closed")
2. When a command is received (via MQTT or REST), the state is updated
3. A timestamp is recorded for the state change
4. The updated state is reported to the Resource Catalog

3. MQTT Communication Function

From the PDF: "This data is published to the Message Broker via MQTT, enabling real-time monitoring of pipeline health and safety."

Implementation:

- The `MQTTService` class handles all MQTT communication
- Connection is established in `connect()` method
- Data publishing occurs through `publish_temperature()` and `publish_pressure()` methods
- Valve commands are received through subscription to the valve topic in `on_connect()`

Code Process:

1. The service connects to the MQTT broker specified in configuration
2. It subscribes to the valve control topic
3. It publishes sensor data to the temperature and pressure topics
4. Incoming messages are processed in the `on_message()` callback

5. Reconnection is automatically handled if the broker connection is lost

4. Catalog Update Function

The PDF mentions: "Additionally, it updates the Catalog with sensor and actuator statuses over REST, providing system-wide visibility."

Implementation:

- The `CatalogManager` class handles all interaction with the Resource Catalog
- The `update_sensor_status()` method sends sensor readings to the Catalog
- The `update_actuator_status()` method sends valve state to the Catalog

Code Process:

1. After registration, the service receives a unique ID from the Catalog
2. Current sensor readings are periodically sent to the Catalog via REST
3. Valve state changes are reported to the Catalog in real-time
4. The service uses proper error handling to manage connection issues

5. Configuration Retrieval Function

According to the PDF: "It can also retrieve the configuration about the frequency of data publication, and threshold limits from the Catalog."

Implementation:

- The `CatalogManager.get_configuration()` method retrieves settings from the Catalog
- The `SensorService.update_config()` method applies these settings
- Configuration checks run periodically in a background thread

Code Process:

1. On startup, the service requests its configuration from the Resource Catalog
2. Parameters like publication frequency, temperature mean/std, and pressure mean/std are applied
3. A background thread periodically checks for configuration updates
4. When new settings are detected, they are immediately applied without service restart

6. Control Command Reception Function

The PDF states: "The connector receives control commands from the Control Center (MQTT), allowing for automatic or manual valve operations based on safety parameters."

Implementation:

- The `MQTTService` subscribes to the valve control topic
- The `on_message()` method processes incoming commands
- Commands are forwarded to the `ActuatorService.set_valve_state()` method

Code Process:

1. The service subscribes to the `/actuator/valve` topic
2. When a message arrives, it's parsed as JSON
3. The "command" field is extracted and validated
4. Valid commands ("open" or "closed") are forwarded to the Actuator Service
5. The valve state is updated and the change is logged

7. REST API Function

While not explicitly mentioned as a standalone function in the PDF, the implementation provides a REST API for system integration:

Implementation:

- The `RaspberryPiAPI` class sets up a CherryPy web server
- `SensorResource` exposes the `/api/sensors` endpoint
- `ActuatorResource` exposes the `/api/actuator/valve` endpoint

Code Process:

1. The API server listens on the configured port
2. GET requests to `/api/sensors` return current sensor readings
3. GET requests to `/api/actuator/valve` return current valve state
4. POST requests to `/api/actuator/valve` with a JSON body can change the valve state
5. All responses are properly formatted JSON with appropriate HTTP status codes

Integration with Other System Components

The Raspberry Pi Connector interacts with several other components in the Smart IoT Bolt system:

1. **Message Broker**: Publishes sensor data and subscribes to valve control commands via MQTT
2. **Resource Catalog**: Registers itself, updates status, and retrieves configuration via REST
3. **Control Center**: Receives valve control commands from this component via MQTT
4. **Time Series DB Connector**: Indirectly provides data to this component through the Message Broker

Data Flow

1. **Sensor Data Flow**:

```
Sensor Simulation → MQTT Service → Message Broker → Subscribers (Time Series DB Con
```

2. **Actuator Control Flow**:

```
Control Center → MQTT Message → MQTT Service → Actuator Service → Valve (simulated)
```

3. **Configuration Flow**:

```
Resource Catalog → Catalog Manager → Sensor Service → Data Generation Parameters
```

4. **Status Update Flow**:

```
Sensor & Actuator Services → Catalog Manager → Resource Catalog → System-wide Visib
```

Key Features and Capabilities

1. **Dynamic Configuration**: All parameters can be modified through the Resource Catalog
2. **Fault Tolerance**: Automatically reconnects to MQTT and retries catalog registration
3. **Realistic Simulation**: Uses Gaussian distribution for realistic sensor data
4. **Dual Control Interfaces**: Supports both MQTT and REST for valve control
5. **Comprehensive Logging**: Provides detailed logging for monitoring and debugging
6. **Graceful Shutdown**: Properly closes connections and stops threads when terminated

Conclusion

The Raspberry Pi Connector implementation fulfills all the requirements specified in the project documentation. It provides a reliable interface between the physical components (sensors and actuators) and the broader Smart IoT Bolt system. Its modular design ensures each aspect of functionality is properly encapsulated, making the code maintainable and extensible.