Università
degli Studi
di Messina

Master's Degree in
Engineering in Computer Science

Industrial IoT

Device Registration and Management in Kubernetes Using Custom Operator and CRD

Student:
Antonio Edoardo Ciliberto 575115

Academic Year 2024/2025

Contents

1 Executive Summary

This paper describes the design, implementation and challenges faced in developing a system for recording and managing the lifecycle of external devices (Edge/IoT) within a Kubernetes cluster. The project simulates a realistic production environment by decoupling the request entry point (a Gateway) from the business logic (a Kubernetes Operator). The architecture implements a secure registration workflow, where new devices can only be registered when an administrator explicitly enables a cluster-wide "Pairing Mode". The system was developed in Go and Rust, using cloud-native technologies such as Custom Resource Definitions (CRD), Docker and k3d for local simulation. The paper analyses the architecture of the components, the end-to-end workflow, the technical challenges encountered during development and their solutions, providing a comprehensive overview of the work performed.

2 Introduction and Project Goals

2.1 Context

In modern distributed systems, especially in the Internet of Things (IoT) and Edge Computing, the secure and automated management of large numbers of heterogeneous devices is a critical challenge. Kubernetes, with its declarative model and extensibility, is emerging as an ideal platform to orchestrate not only software workloads, but also external physical devices.

2.2 Goals

The main objectives of this project were:

1. Implement the Pattern a Operator: Develop a Kubernetes Operator in Go to encapsulate the business logic related to the registration and lifecycle of devices.
2. Extending the Kubernetes API: Define a Custom Resource Definition (CRD) DeviceRegistration to declaratively represent the state of a device in the cluster.
3. Simulate a Secure Registration Workflow: Create a system in which the registration of new devices is centrally controlled via a "Pairing Mode", which is disabled by default.
4. Decoupling Access from Logic: Realise an HTTP Gateway as a single entry point for devices, separating it clearly from the approval logic managed by the Operator.
5. Simulate a Real Device (MCU): Develop a client (in Rust) that simulates the behaviour of a microcontroller (MCU) that autonomously initiates the registration request.
6. Create a Complete Test Environment: Use k3d (Kubernetes in Docker) to create a complete local environment in which all components could be deployed and tested.

3 System Architecture

The system consists of several micro-services and Kubernetes components that interact to realise the workflow.

3.1 Custom Resource Definition (CRD)

The heart of the data model. This resource extends the Kubernetes API and defines the structure of a registration request:

- spec: Contains the data provided by the requester (the Gateway), such as the publicKey of the device. It also contains a deactivate flag for lifecycle management.
- status: This is the field managed exclusively by the Operator. It contains the outcome of the process:
  - phase: The current status (Pending, Approved, Rejected, Deactivated).
  - message: A readable message describing the status.
  - deviceUUID: The unique identifier assigned by the operator in case of approval.

3.2 Registration Gateway (MPU)

A lightweight web service written in Go, acting as a Micro-Processing Unit (MPU) or input proxy.

- Responsibilities: Exposes an HTTP endpoint (POST /enroll) and receives requests from devices. It translates these requests into the creation of a DeviceRegistration resource in the Kubernetes cluster.
- Logic: After creating the resource, the Gateway enters a polling loop, checking the status of the resource until the Operator processes it. Finally, it returns the outcome (the UUID in case of success, an error in case of failure) to the device.
- Security: Acts as a "bastion", preventing external devices from communicating directly with the Kubernetes server API.

3.3 The Operator

The core component, written in Go with the Kubebuilder framework.

- Responsibility: Contains all business logic.
- Logic:
  1. Monitor the creation of new DeviceRegistration resources.
  2. When one is detected, it checks the status of the ConfigMap.
  3. If pairing is enabled (enabled: "true"), it approves the request, generates a UUID and updates the status of the CRD.
  4. If pairing is disabled, it rejects the request and updates the status with an error message. It also handles deactivation requests.

3.4 Device Simulator (MCU)

A command-line script written in Rust, which simulates the behaviour of a Micro-Controller Unit (MCU).

- Responsibility: Start the registration process.
- Logic:
  1. Generates a dummy public key.
  2. Prepare a JSON payload.
  3. It makes an HTTP POST request to the Gateway's /enroll endpoint.
  4. It interprets the Gateway's response and shows it to the user.

4 Workflow End-to-End

The complete process of registering a new device follows these steps:

1. Enabling (Administrator): A system administrator enables the registration of new devices by editing the ConfigMap.
2. Request (MCU): The Rust client is executed. It generates a public key and sends a POST request with the key in the JSON body.
3. Resource Creation (Gateway): The Gateway receives the request. It authenticates with the Kubernetes API and creates a new DeviceRegistration resource in the namespace 'device-operator-system'. The resource name is generated randomly.
4. Reconciliation (Operator): The Operator, which monitors resources, detects the new DeviceRegistration. It starts its reconciliation cycle.
5. Validation (Operator): The Operator reads the pairing ConfigMap. Since it is enabled, it decides to approve the request.
6. Status Update (Operator): The Operator generates a unique UUID, sets "status.phase" to "approved", and updates the resource in the cluster.
7. Polling (Gateway): The Gateway, which was checking the status of the resource at regular intervals, detects that the phase has changed to Approved. It reads the deviceUUID from the status.
8. Response (Gateway): The Gateway sends a 200 OK response to the MCU client, including the UUID in the JSON body.
9. Confirmation (MCU): The Rust client receives the success response and prints the received UUID on the screen, completing the cycle. If the pairing mode had been disabled, in step 5 the Operator would have set the phase to Rejected, and the Gateway would have returned a 403 Forbidden error to the client.

5 Challenges and Solutions

Several technical challenges arose during development, the resolution of which was crucial to the success of the project.

1. Networking problems with k3d: The curl client (and then the Rust client) could not connect to the gateway on port 30007, returning Failed to connect.
    - Cause: On macOS, Docker runs in a VM. A service of type NodePort is only exposed within the Docker network, not on the localhost of the host machine.
    - Solution: Recreate the k3d cluster using the port-forwarding flag: k3d cluster create dev -p "30007:30007@loadbalancer".
2. kubectl's NotFound Error: After correcting the problems with the CRD, kubectl continued to search for the resource with the wrong API name.
    - Cause: The local kubectl cache (/.kube/cache) still contained the information of the old and incorrect CRD definition.
    - Solution: Clear the kubectl cache with "rm -rf /.kube/cache/discovery" or use the full resource name "(deviceregistration.devices.example.com)" to bypass the short name cache.

6 Tech Stack

- Languages: Go, Rust
- Orchestrator: Kubernetes
- Local Environment: k3d (Kubernetes in Docker), Docker Desktop
- Framework Operator: Kubebuilder / Operator SDK
- Go Main Libraries: client-go, controller-runtime
- Rust Main Libraries: tokio, reqwest, serde
- Build and Automation: Make, Cargo

7 Conclusions

The project successfully achieved all its objectives, producing a working system that demonstrates a robust and scalable device registration pattern on Kubernetes. The challenges encountered provided valuable practical lessons on debugging cloud-native applications, network configuration and resource lifecycle management in Kubernetes.