Enabling QoS-Aware Task Execution on Distributed Node-RED Cluster for Fog Computing Environments

First milestone

Content

1. Project status
2. Faced issues
3. Next steps

Project status: Fog node / node-RED infrastructure

- Created docker images intended to run on fog nodes (x86 / arm)
  - Contains node-RED software and additional packages / node-RED nodes
  - Setup initial flows that are necessary to connect to the fog infrastructure
  - Connects to fog orchestrator (DNR editor) and mqtt server after startup (addresses are defined via environment variables)
  - Periodically sends a heartbeat including the current system load
  - Able to receive shell commands via MQTT, execute them, and return the result
- Created docker image running the DNR editor
  - Necessary for fog nodes to be able to connect to the DNR editor
  - Flows can be orchestrated at this central location and deployed to the all connected fog nodes
- Set up testbed containing three nodes
  - Fog orchestrator (MacBook Pro) running DNR editor, MQTT Server, Monitor
    - Can also behave as a third fog node
  - 2x Fog nodes (Raspberry Pi 3B+, Raspberry Pi 4B) able to run docker containers
  - All devices connected via Ethernet / WiFi to the same router

Project status: Fog orchestrator

- Setup Maven project containing necessary dependencies (JSON / MQTT)
- Implemented monitor which gathers information about the current fog network state
  - Connects to the MQTT broker, receives heartbeats from fog nodes
  - Parses heartbeat JSON objects into Java objects
  - Saves / updates those values in a HashMap
- Prepared HTTP request to the (distributed) node-RED REST API

Project status: Object detection

- Got familiar with the basics of object detection, TensorFlow, OpenCV and ImageAI (completely new field)
- Setup object detection environment on Mac, Raspbian and (finally) Docker
  - *While the setup on macOS (x86) was very quick and easy, it took a long time for Raspbian (arm) because additional dependencies are needed. Raspbian is currently not as well supported as Mac / Linux / Windows.*
  - *Getting all dependencies required for ARM into a docker image took even more time*
- Wrote sample scripts which are able to detect objects in images, videos or camera streams
- Implemented REST webserver which processes images
  - Unprocessed image is sent via POST request (raw image/jpeg data)
  - Processed image is returned (raw image/jpeg data)
- Created a node-RED flow which takes a picture from webcam, sends it to the REST API and saves the response (processed image) to disk

Project status: Sensor data

- Examined some existing sample data sets
- Nothing implemented yet

Faced issue: distributed node-RED

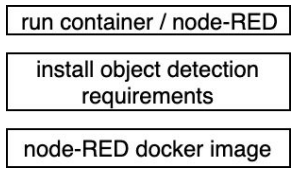
- Suitable for sending (small) JSON messages between nodes, not serialized data like raw image data / streams required for object detection
- Not developed very actively
  - Not very stable and crashes occasionally
  - I even had to fix a bug myself to get it to run inside a docker container at all
- Initial idea: Orchestrate flows entirely in DNR editor
  - 

- New idea: Manage communication between nodes without DNR
  - using available core nodes like TCP, WebSocket, HTTP, MQTT

Faced issue: Running object detection / python code inside node-RED

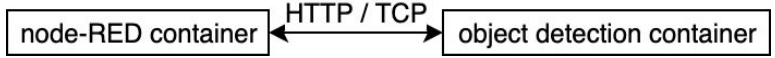
Initial idea: Run object detection inside node-RED (same docker container)

- Problem 1: Initialization time
  - Object detection needs to be initialized before the first image can be processed
  - → Should be done only once upon startup
  - Executing (and initializing) the script for every picture is not efficient
- Problem 2: Passing data (image / stream)
  - Only possible via STDIN, but then again, the script has to be initialized every time



New idea: Separate, dedicated docker image for object detection

- Contains python and all dependencies required for object detection
- Initialized only once
- Communication via HTTP / TCP / WebSocket (available as core nodes in node-RED)
- Can even run on the same host (node)
- node-RED docker image will not be extended → clean docker images



Faced issue: ImageAI not working with Flask

- First, object detection was realized with ImageAI
  - is based on TensorFlow and OpenCV
  - abstracts the TensorFlow object detection API to realize object detection using just a few lines of code
- When implementing the web service using the python Flask framework, ImageAI crashed during processing (error message from TensorFlow)
  - Was not able to quickly fix it because ImageAI is too abstract
- Solution: Object detection is now implemented without ImageAI (just using TensorFlow) → more lines of code, less abstract
- Next problem: Processing time (see next slide)

Faced issue: Object detection execution time

- Acceptable initialization time (dataset loading time)
  - MacBook Pro: ~0.5 seconds
  - Raspberry Pi 4B: ~8.5 seconds
- Relatively long processing time per image
  - MacBook Pro: ~2.5 seconds
  - Raspberry Pi 4B: ~10.8 seconds
- Times are based on the current object detection implementation using TensorFlow
- ImageAI in combination with different datasets was faster as far as i can remember (could be investigated if ImageAI would still be an option)

Faced issue: Distribute object detection from camera

- A single image can easily be sent, (externally) processed and received via a single HTTP request and response
- On a single device, the camera stream can easily be forwarded directly to the object detection engine
- Sending, processing (externally), receiving and displaying the camera processed stream, on the other hand, is a challenge that has yet to be solved.
    - Idea 1: Send and receive camera stream via TCP
    - Idea 2: Communication via RTSP streams

Next steps

1. Object detection
    - Optimize image processing time
    - Handle camera streams (required?)
2. Sensor data
    - Define a task which will be executed using the collected sensor data.
    - Choose a suitable sample dataset for that task
    - Create a flow in node-RED
3. Fog orchestrator
    - Implement MAPE-K
        - Use the data (**K**nowledge base) collected by the **M**onitor for optimal flow distribution
        - **A**nalyze data using FogTorch → Result: optimal distribution **P**lan
        - **E**xecute: Deploy that plan onto the fog nodes