AY 2020/2021

POLITECNICO DI MILANO

Middleware Technologies
Contact Tracing with IoT Devices

Federico Armellini   Luca Pirovano   Nicolò Sonnino

Professor
Luca MOTTOLA

Version 1.0
May 8, 2021

Contents

1 Introduction

## 1.1 Description of the project

People roaming in a given location carry IoT devices.

The devices use the radio as a proximity sensor. Every time two such devices are within the same broadcast domain, that is, at 1-hop distance from each other, the two people wearing the devices are considered to be in contact.

The contacts between people's devices are periodically reported to the backend on the regular Internet. Whenever one device signals an event of interest, every other device that was in contact with the former must be informed.

2 Solution Overview

2.1 General Architecture
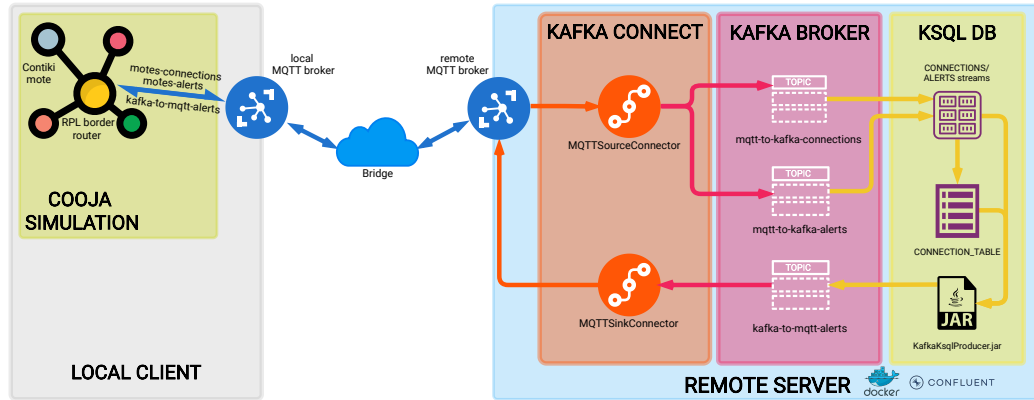
The project consists into an integration of Contiki-NG motes with Apache Kafka using Confluent Platform and Docker.

Confluent was used for an easy management of Apache Kafka while Docker was utilized for deploying Confluent Platorm and Apache Kafka into containers (increasing scalability and manageability of the infrastructure).

Platforms

- **Apache Kafka**
- **Confluent Platform**
- **Contiki-NG**
- **Docker**
- **KSQL**

Protocols

- **MQTT**
- **RPL**
- **UDP**

Components

The following list explains each component of the workflow:

- **Contiki-NG motes**: motes running `contact-tracing.c` program and emulated in the COOJA simulator or compiled natively.

  They use RPL (Routing Protocol for Low-Power and Lossy Networks) protocol in order to exchange messages between them and MQTT (Message Queue Telemetry Transport) protocol for communicating with the middleware. Each mote runs three processes:

  1. <u>Broadcast</u>: this process fires a broadcast UDP message containing the mote's client ID at regular intervals.
  2. <u>Broadcast receiver</u>: process responsible for receiving clients IDs via UDP from other motes and inserting them into the MQTT queue buffer in order to publish them later.
  3. <u>MQTT main process</u>: main process, it manages the whole MQTT workflow: it publishes to `mqtt-to-kafka-connections` and `mqtt-to-kafka-alerts` MQTT topics both the motes' connection between each other and events called alerts at random intervals. It also subscribes to the `kafka-to-mqtt-alerts` topic listening to incoming events.

- **RPL border router**: mote acting as a router for sharing messages between motes and outside connections.

- **MQTT broker**: it is the component handling messages from clients and topics, it is deployed both on the local client and on the remote server. They are both connected: the local one uses bridge mode and forwards every MQTT event to the remote one (very useful when using IPv6 or if the local client is running into a Virtual Machine).

- **MQTTSourceConnector**: Confluent connector (using Kafka Connect functionality) which reads from a MQTT topic (connection and alert in this case) and writes each publish on a Kafka one.

- **Kafka broker**: broker managing Kafka topics and connectors and interfacing with the KSQL system.

- **KSQLdb**: KSQL is a technology developed by Confluent, it allows to keep Kafka streaming versatility and it combines with a SQL language. It implements two different structures:

  1. <u>Streams</u>: continuous queries of streams of data in Kafka topics, they insert the content into the KSQLdb in order to store data. In the implementation two streams were instantiated: one for connections and one for alerts.

  2. <u>Tables</u>: tables are views of streams and represents a collection of evolving facts. Every column specified into the table has a related value from the stream/table used for its creation. The main advantage of using KSQL tables is that updated values coming from streams are overwritten over the previous ones effectively avoiding duplications or outdated values. A table is used for storing connections between a node and the other ones.

- **KafkaKsqlProducer.jar**: Java program which creates two streams (CONNECTIONS and ALERTS), defines a table (CONNECTION_TABLE) and instantiates a Kafka producer. Whenever an alert is received into the stream, the jar queries the CONNECTION_TABLE (continuously updated by the CONNECTION stream) with the client ID contained into the message. The results are client IDs connected previously with that mote and they are used for each ProducerRecord wrote by the Kafka producer and sent to the "kafka-to-mqtt-alerts".

3

- **MQTTSinkConnector**: Equivalent of the MQTTSourceConnector, it receives messages from a Kafka topic and writes on a MQTT one.

2.2 Flow of execution

2.3 Assumptions

A1: The IoT devices may be assumed to be constantly reachable, possibly across multiple hops, from a single static IoT device that acts as a IPv6 border router, that is, you don't need to consider cases of network partitions.

A2: The IoT part may be developed and tested entirely using the COOJA simulator. To simulate mobility, you may simply move around nodes manually.

A3: Notification at a target device may be accomplished in simple ways, for example, by turning on a LED or printing something out on the serial console.

A4: Two IoT devices had a contact if they were reachable at 1-hop distance during a whole minute.