AY 2020/2021

Politecnico di Milano

Middleware Technologies
Contact Tracing with IoT Devices

Federico Armellini Luca Pirovano Nicolò Sonnino

Professor
Luca Mottola

Version 1.1
September 2, 2021

Contents

1 Introduction

1.1 Description of the project

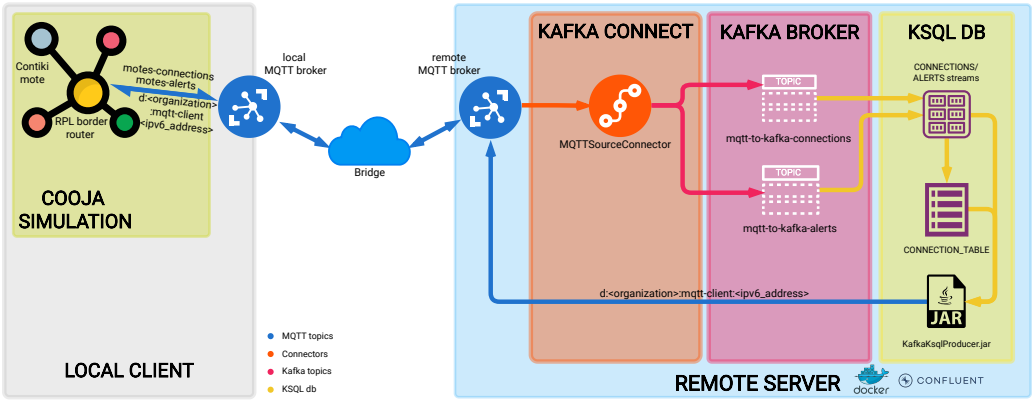
People roaming in a given location carry IoT devices.

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. Broadcast: this process fires a broadcast UDP message containing the mote's client ID at regular intervals.
  2. Broadcast receiver: 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. MQTT main process: 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 its alert topic, which is simply the ID of the device, 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. Streams: 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. Tables: 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 single topics of interest of the motes who has been in contact with it.

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

2.2 Flow of execution

With reference to General Architecture (figure in paragraph 2.1) the execution flow is divided as follows:

- At the beginning of the execution of the Cooja simulation, a random timer is set for each single client mote to communicate the alerts (alerts will be sent to report Covid19 positivity).
- The individual motes, in order to exchange the actual contact information among themselves, communicate with each other at regular intervals via UDP broadcast, sending their own id. Each mote, upon receipt of one of these packets, puts each of them in a "contact" queue, which gradually fills up to be sent to the server later.
- Each motes (at regular intervals) notifies the server (via mqtt in the topic "motes-connections"), telling it about the contact between those two by referring to the previously filled queue: the message sent is composed of the id of the sender mote and the id of the mote wich has been in touch with the sender. After sending, the queue is emptied. The connection between the two motes is recorded in the KSQL “CONNECTIONS” table. For this type of message, the QoS chosen in use by the mqtt publish is 1: in a delivery and traffic guarantee tradeoff we preferred to opt to preserve a correct execution of the system, as contact is essential to then correctly determine the history. of the contacts of a possible positive.
- Each motes at random intervals notifies the server to be positive, thus sending its id inside the message. (topic mqtt-alerts). The positivity information, called "alert", is saved in the KSQL "ALERTS" table. For this type of message, the QoS chosen in use by publish mqtt is 1: as for the previous message, also for the notification of positivity it was considered essential by us to guarantee its correct transmission, in order to carry out the simulation correctly.

- In the jar producer "KafkaKsqlProducer.jar" residing in the server, in polling state on the KSQL “ALERTS” table, when it notices a positivity, it makes a query to KSQL on the CONNECTIONS table, filtering for the infected client's contacts. For each of the infected client's contacts the server sends a dedicated notification via mqtt (via a topic dedicated to each "d: CLIENTID" mote). The jar server also takes care of possibly recreating the tables and database structure, should they not be present. For this type of message, the QoS chosen in use by the mqtt publish is 1 for the same reasons described in the previous messages.

Each mote during its execution prints on the screen in the cooja simulation all the messages received.

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.