Politenico di Milano

Middleware Technologies for Distributed Systems

Project Report

Contact Tracing with Body-worn IoT Devices

Author:
Fabio Losavio
Matteo Velati
Niccolò Zangrando

Supervisor:
Dr. Luca Mottola

April 11, 2021

1 Introduction

1.1 Description of the given problem

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.

1.2 Assumptions and Guidelines

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

1.3 Technologies

The technologies involved in the development of this project are Node-RED and Contiki-NG.

2 Design and Implementation

2.1 COOJA

The whole project is structured with a event-driven programming pattern, where the motes are the objects responsible for triggering such events. The wireless propagation model we chose for our simulation is the *Constant Loss Unit-Disk Graph Model (CL-UDGM)* with a 0 meters interference range, meaning that the communication between motes in the same communication range is fully reliable.
Based on the *Routing Protocol for Low-Power and Lossy Networks (RPL)*, all the motes in the network got arranged in tree-shaped topology rooted at the node with direct Internet access and communications among those exploits the UDP protocol.
Moreover, we decided to use the COOJA motes because of the lack of limitations on memory-size of the software to be loaded on them. The are three different motes involved in the simulation: a RPL BORDER-ROUTER, a MQTT-UDP MOTE and a UDP SIGNALER.

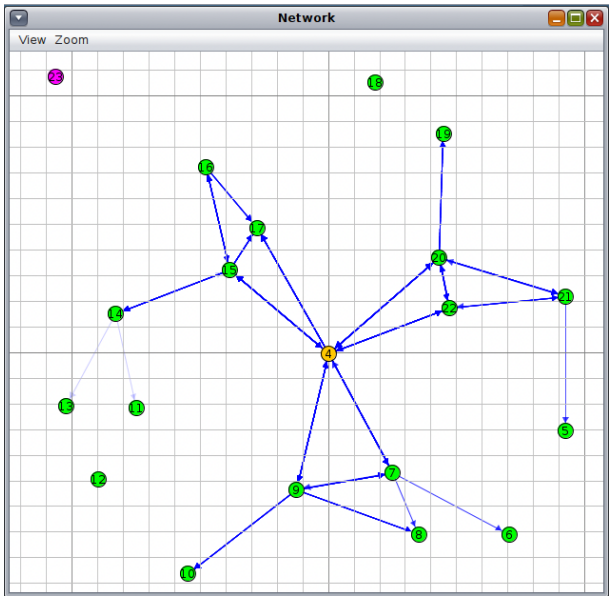

Figure 1: *The IoT network*

2.1.1 RPL Border-Router

The RPL BORDER-ROUTER mote is a specific IoT device that acts as a *man in the middle* to bridge the wireless sensor network with the external internet. It is configured as the *Destination-Oriented Directed Acyclic Graph (DODAG)* root of the whole network, and the it consists of a single thread which allow communication with the external internet.

2.1.2 MQTT-UDP mote

This kind mote is the core of our project; it exploits protothreads to achieve parallel execution of UDP and MQTT communication.
The MQTT process consist of a *Finite State Machine (FSM)* which is periodically triggered by a timer and takes care of handling MQTT events as well as connecting the local Mosquitto broker, that relies on the virtual machine, to the one hosted on the internet: while doing so, the tree-topology structure is created since messages are redirected between each mote to the RPL BORDER-ROUTER which is the only one which actually has internet access.

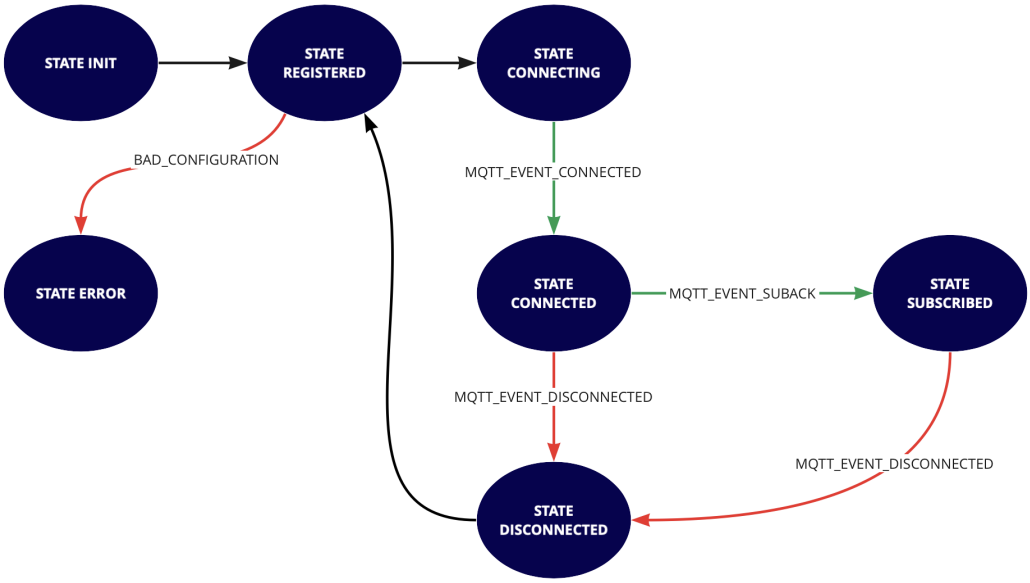

Figure 2: *MQTT finite state machine*

MQTT communication follow the publish and subscribe messaging pattern: each mote subscribe to a topic that is built at run-time by the concatenation of the fixed string *mdw2021/notifications/* with the IP address of the mote itself. The *Quality of Service (QoS)* chosen is 0 in order to minimize the overhead on each mote. The subscription to a different topic for each mote allows the back-end to properly communicate with them separately: whenever a mote M1 reports an event of interest, the back-end sends to every mote on the contacts list of M1 a message containing its IP address; then each contacted mote prints on the console a notification message. The purpose of these messages is to signal that a mote with which it was previously in contact has now reported such an event of interest.
Each mote has two publish topics built on the IP address too, one for `contacts` and one for `signals`, used for communication with the back-end. In this case the *QoS* is set to 1, with a *at least once* semantic, since the backend can handle the duplicates.
Regarding the UDP communication process, each mote is both client and server: as a client, it periodically multicast a simple message to each mote available in its neighbors list; as a server, it receives those kind of messages which triggers a callback function. The callback function publishes on the `contacts` topic a *JSON* message containing the two IP addresses, so that the back-end can store a permanent list of contacts of each mote.

2.1.3 UDP signaler

Whenever one device signals an event of interest, every other device that was in contact with the former must be informed. In order to achieve this functionality, we created a special mote that keeps multicasting a different message to every mote in its neighbors list, if any. Each MQTT-UDP MOTE, upon receiving that particular message, publishes on the `signals` topic its own IP address in a *JSON* format, so that the back-end can alert the motes on its contacts list of the report.

2.2 Node-RED

The technology we chose for back-end processing is Node-RED, which stores data on Google Firebase realtime database.
There are two event-driven flows and their initial work is basically the same: the very first node, a MQTT receiver, take the advantage of the multi level wildcard #, which allows to receive all messages with additional topic levels; the messages are then parsed from *JSON* to *JavaScript* objects and processed once at time, while the others remain enqueued until all database called are performed.
The MQTT *QoS* chosen for both subscribing and publishing events is 1, which guarantees that a message is delivered at least one time to the receiver.

Signals flow The messages received on the `signals` consist of a single IP address: a call to the database is performed to retrieve all the contacts stored for that IP address. Then, for each pulled contact, a message on its topic is being published, containing the identifier of the node which is sending the notification. Once finished the publication, all the key-value entries in database containing the IP address received are deleted, and the next enqueued message starts being processed.

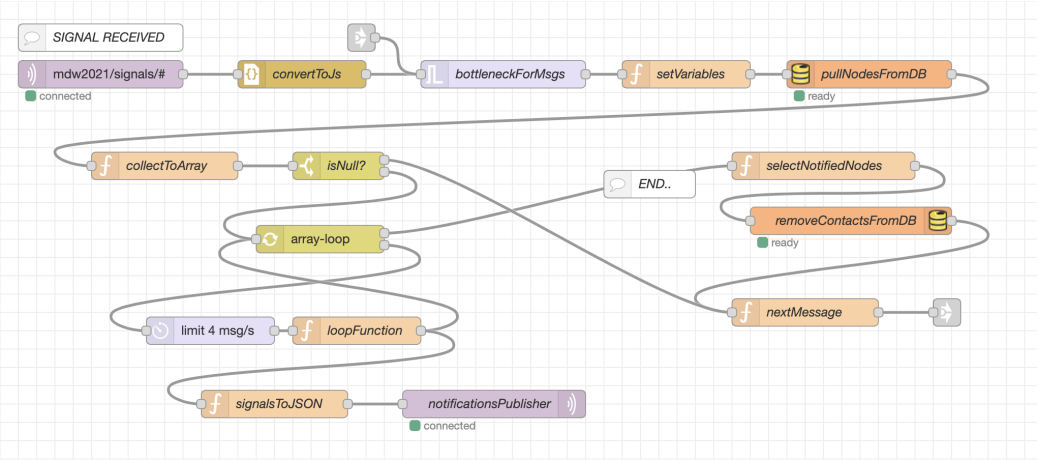

Figure 3: *Flow management for all signals topics*

Contacts flow Each message received on the `contacts` topic is a contact between two motes; we check if the contact isn't present in the database, and if so, we add the two entries `IPx -> IPy` and `IPy -> IPx`. After that, processing of the next queued message begins.

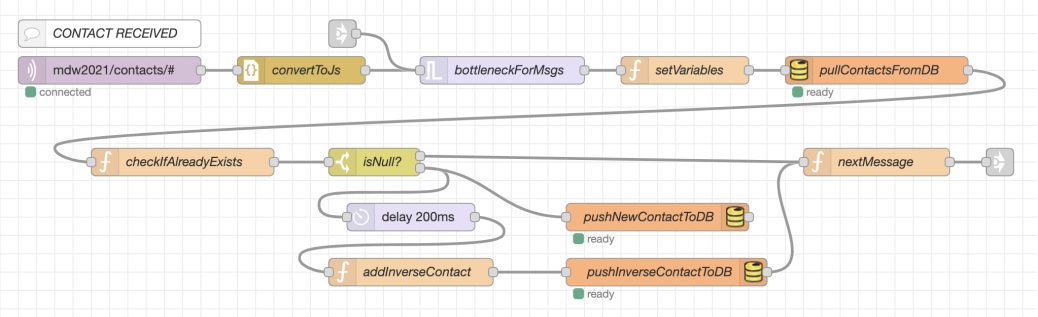

Figure 4: *Flow management for all contacts topics*

Database structure We use the NoSQL realtime database from Google to store our values after processing them. A two-levels *JSON* tree structure is created, with each node at the first level and contacts and the seconds level.

Figure 5: *Google Firebase NoSQL database structure*

3 Experimental results

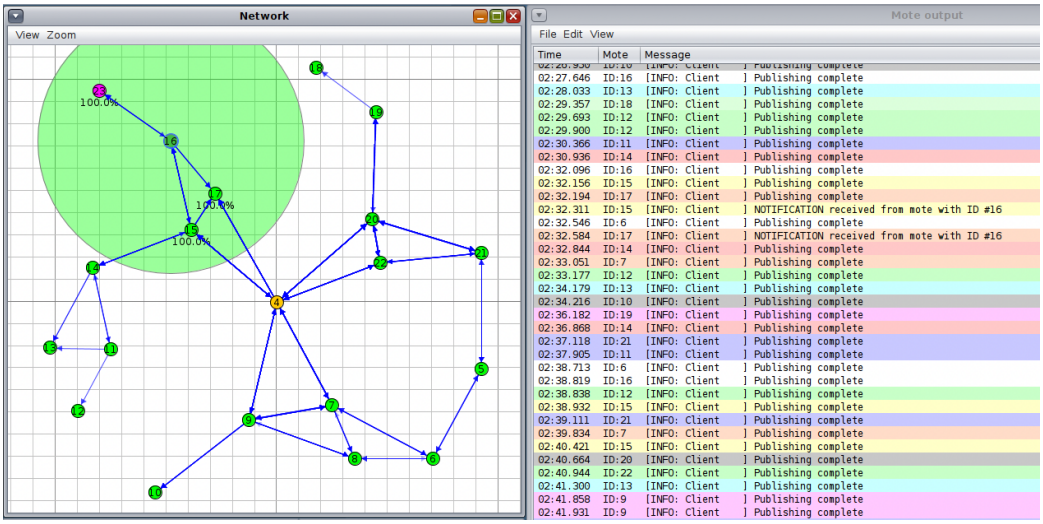

Figure 6: *The simulation running inside COOJA environment*

In the Figure above, we can see how the interactions among those motes is actually performed: after some time, the tree-shaped topology has been created, each mote is registered and connected to the MQTT broker and they are exchanging messages with the motes in their neighborhood. Each time a mote receive a message, it publish it to the back-end and prints on the console the message *Publishing complete*. After some time, the contacts lists start to be filled with IP addresses, and if we bring the UDP SIGNALER mote (the purple one) enough close to another mote, we trigger the event of interest.
In this particular case, it's the mote number 16 which sent the signal to the back-end: the contacts list it's downloaded and erased, plus a notification message is sent to each entry of that list, which in this case consists of motes 15 and 17. Both motes prints on the console inside the simulation that they have received that notification from the mote which starts the process of reporting.

4 References

- Contiki-NG documentation:
  https://github.com/contiki-ng/contiki-ng/wiki/
- Node-RED documentation:
  https://nodered.org/docs/
- MQTT documentation:
  https://www.hivemq.com/mqtt-protocol/