A Faulty IoT Network: Simulating Sensors and Perturbations

Kenza Riahi[1], Giacomo Kahn[1] Baudouin Dafflon[2], and Jannik Laval[1]

[1] *DISP-LAB, Univ Lyon, Univ Lumière Lyon 2*
Lyon, France, firstname.lastname@univ-lyon2.fr
[2] *DISP-LAB, Univ Lyon, Université Claude Bernard Lyon 1*
Lyon, France, baudouin.dafflon@univ-lyon1.fr

Abstract. A simulated system has certain advantages over a physical one. It can be cheaper to implement, destroyed with little to no consequences and more generally toyed with basically no cost. To test fault detection algorithms, there is a need for the simulation of a faulty IoT network. In this paper, we present a platform for simulating IoT exchanges, where sensors can be real or virtual, only editors or both editors and consumers. The platform allows defining a wide variety of simulation parameters such as the number of sensors to instantiate, each sensor settings, perturbations to inject and stop conditions. We use the RabbitMQ broker for message exchanging and the Open Data Format for standardizing messages format. We detail the global architecture and present two case studies to show how the platform works. We build a "smart" sensor that retrieves measures from other sensors and uses them to predict the temperature value based on an artificial intelligence model and show that any model can be instantiated as a smart sensor.

Keywords: IoT exchanges, simulation, modeling, scenario, configuration.

1 Introduction

Data exchanges within IoT systems should be continuously monitored in order to evaluate their performance and detect failures as soon as they happen. An IoT system is composed of interconnected "things", generally sensors and actuators, that exchange data. Sensors capture data from the real surroundings and send a periodic measure (e.g. image, temperature, position). Their main role is to control the environment and prevent risks. The actuators receive orders and apply them in the real world. In this work, we only focus on sensors.
The implementation of monitoring systems requires to collect the exchanges within the IoT system, a task that can be delicate or even impossible when data is sensitive or irretrievable. Simulation, in this case, can facilitate the generation of these exchanges for a better exploitation. Another point is the composition of the simulated system: testing large systems with only real sensors results in high costs or in the difficulty to

capture some measures. Virtual sensors can be used to generate more or less realistic data and send them to the system.
We propose in this paper a simulation platform. It uses a RabbitMQ broker for retrieving the measurements from the sensors. It is used with the MQTT protocol and each message is sent to the appropriate queue. This point is interesting for this work because we set up virtual sensors that are both consumers and editors: they get data from the broker and use them to generate the target measure that is sent to a specific queue.
The core of our simulator is a configuration file that enables defining the scenario to be launched in all its details (number and type of sensors used, the perturbations to inject, stop conditions, etc.). Once the user fills it in, they can run the simulation and monitor it until its end when the simulation metrics are calculated by the platform.
The format of exchanged data is also important to ensure interoperability within the system. We use the Open Data Format (O-DF) [3] specified by The Open Group Internet of Things Standards and based on the XML scheme.
The implemented platform is based on a generic model that can easily be extended. It defines the main simulation components: sensors, filters, stop condition for each sensor, stop condition for the whole simulation and returned metrics at the end of the run.
We show the usability and the extensibility of the model by applying it to two case studies. The first one runs a number of virtual sensors simultaneously with specific parameters and gets the metrics at the end of the simulation. The second case study introduces a typical temperature sensor that uses machine learning to predict measures on the basis of the retrieved values of air density, humidity and pressure from the broker.
This paper is structured as follows: Section 2 presents the related work, Section 3 details the structure of the simulation platform. In Section 4, the two case studies are presented and evaluated. Finally, we conclude our work, in Section 5, with some perspectives.

2 Related work

Simulation tools in the IoT area have gained a lot of interest due to their importance in testing systems before their deployment or difficult scenarios in the physical world.
A number of articles established a comparison between the existing IoT simulators, for example the survey by Singh, Vyas and Tiwari [9], the one published by Saidallah, El Fergougui and Elbelrhiti Elalaoui [8] or the review conducted by Bakare and Enoch [1]. In [2], simulators are divided into three categories: Full Stack Simulators, Big Data Processing Simulators and Network Simulators. Only the third one is interesting in the case of our study since we only need to simulate data exchanges within an IoT system regardless to the data size and with no need to process the exchanged data.
We briefly present the most popular and open-source tools in this category which are used for discrete event simulation.

[3] `https://publications.opengroup.org/c14a`

NS-2 [6, 3] and its successor NS-3 are object-oriented discrete event simulators, extensible thanks to their modular approach. However, they are not scalable, lack available customization and have low computational overhead.
OMNET++ [5, 12, 11] is another simulator that has an architecture based on modules which make it extensible. However, it is not widely used by the community.
J-Sim [10] is a Java-based simulator that has a component-based architecture which gives the tool many advantages in terms of scalability. However, it is relatively complicated to use.
SensorSim [7] enables hybrid simulation by integrating both hardware and software components and offers graphical data display.
In [4], a platform is developed for simulating sensors in the context of a factory to prevent risks that can be encountered by the workers. It enables to generate realistic and real time observations, it uses a Sensor Observation Service (SOS) server for storing sensor measurements.
All these simulators are interesting, but none of them includes all the functionalities we need in our study: the possibility to inject perturbations in the system and the need for an extensible model of these perturbations, in addition to the output metrics we need to evaluate the system.

3 Simulation platform

3.1 Architecture

The platform is based on the class diagram of Figure 1. It represents all the implemented classes used to run the simulation.
Our model is composed of the following classes:

- Class `Simulation` reads the configuration file to launch the simulation with the defined parameters and sensors. It returns metrics at the end of the run.
- The abstract class `Sensor` allows instantiating a sensor by defining a number of attributes: name, measurement type (a temperature, a pressure, etc.) , data type (double, integer, etc.), measurement unit, sensor location, measurement interval, sending frequency and sending routing key so that the runSensor method knows to which queue the message will be sent.
  Some information is useful for the sensor operating and others for the content of the message sent to the broker.
  This class is abstract so that it can be generic and the user can create a sensor that works in any way. In fact, we defined two sensor types: sensors that generate a message and send it to the broker and others that consume messages and use them to generate the measure.
  In the class diagram of Figure 1, two classes inherit from `Sensor`: `RandomNumericalSensor` that generates a random value from the measurement interval and sends it to the broker, and a second class `EditorConsumerSensor` that is abstract. This second class retrieves values from the broker and uses them to generate the measurement. The class `TemperaturePredictor` inherits from it.

- The abstract class `Perturbation` generates perturbations during the simulation. It has three child classes: `DeleteDataPerturbation` class that deletes the measurement, and thus it is not sent to the broker, `AlterDataPerturbation` class that changes the measurement value before it is sent to the broker and `TemporarySensorStop` for stopping the sensor temporarily.
- The abstract class `Filter` allows using filters before sending a value to the broker. We defined two child classes: `LowPassFilter` class that fixes a maximum value for the measurements and `HighPassFilter` class for defining a minimum value. Filters adjust the measurement if it doesn't respect the defined maximum and/or minimum values for the filters.
- The abstract class `SensorFinalStop` for defining the sensor's stop conditions. Two child classes were created: `TimeElapsedFinalStop` and `NumberOfMessagesFinalStop` that respectively allow fixing the sensor's operating duration and the number of messages sent before the sensor stops.
- `SimulationFinalStop` is an abstract class that defines stop conditions for the simulation. We implemented two child classes to define a run's duration before it stops and to fix a maximal number of messages sent to the RabbitMQ broker: `TimeElapsedBeforeStop` and `SentMessagesBeforeStop`.

From this model, we implemented Python classes from which a simulation can be run. It is extensible thanks to the abstract classes we created and that can be extended with classes adapted to the user needs.
The platform implements abstract classes that can be subclassed which allows the creation of new functions for the simulator, and concrete classes that implement useful methods for the simulations. These classes and methods retrieve their parameters from a configuration file filled by the user that contains the most important information for each run such as the simulation duration, sensors types, perturbations and stop conditions.
Data exchange uses Message Queuing Telemetry Transport (MQTT) protocol via a RabbitMQ broker. At the end of the simulation, some metrics are calculated and written in a CSV file.
The source code of our model is available at: `https://github.com/disp-lab/IoT-Simulator`.

3.2 Simulation configuration

The model we detailed in the previous section allows defining a configuration file for the simulation. It contains information about the simulation: name, date, stop conditions and the list of the sensors to instantiate. Each sensor should be well defined with all its parameters: type, frequency of sending messages, sending routing key, used filters, injected perturbations and stop conditions for the sensor.
The method `Simulation.runSimulation()` uses this file to run the simulation. It launches all the defined simulators using the specified parameters, uses the stop conditions to stop the simulation, computes the metrics at the end of the simulation and returns the calculated values.

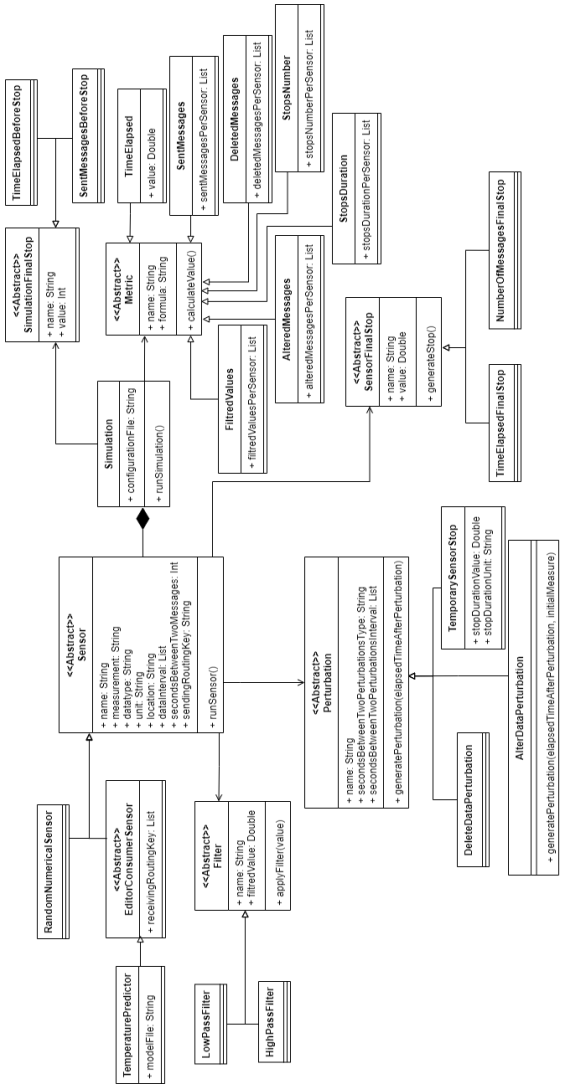

Fig. 1. Architecture of the simulator model.

3.3 Format of exchanged data

For messages exchanging, we defined a standardized data format [4] based on the Open Data Format (O-DF) proposed by the Open Group Internet of Things Standards. It uses the XML scheme and specifies, for each message, information about the component that sends the data and information about the data itself.

[4] `https://github.com/disp-lab/IoT-Simulator/blob/main/sensors/src/messages_format.xml`

3.4 Metrics

Previously, we presented the architecture of our model for launching a simulation after defining all its parameters. We added another abstract class `Metric` that calculates and returns a metric at the end of the run. We defined some classes that inherit from this abstract class: `TimeElapsed`, `SentMessages`, `FiltredValues`, `DeletedMessages`, `StopsNumber`, `StopsDuration` and `AlteredMessages`. Those last four metrics come from the perturbations that can be injected during a sensor operating. `FiltredValues`, however, calculates the number of values that were filtered before sending them to the broker. The values of these metrics are written in a CSV file at the end of the run.

4 Experiments

4.1 Overview

This section presents two scenarios defined for the simulator. The first one is simple and runs a number of sensors in order to prove that the simulator works well. In the second one, we extend sensors to build a virtual sensor that both consumes messages from the broker and sends a calculated temperature value, based on a machine learning algorithm. The computer we used to launch these case studies has an Intel Core i5 7th generation processor with a 2.71 GHz CPU and a RAM memory of 8 Go.

4.2 First case study

In the first case study[5], we run 50 sensors that simultaneously publish messages with random measurements to the RabbitMQ broker. We set the simulation stop condition to 1000 messages sent.
We choose three sensor types: air density sensors, humidity sensors and pressure sensors each one sending messages respectively to the three queues with the routing keys “airdensity”, “humidity” and “pressure”. Fifteen sensors will send messages to the first queue, 15 to the second one and 20 sensors to the third queue.
For each sensor, we choose a data interval and introduce filters and perturbations so that all the implemented classes are tested in this case study.
Table 1 presents the simulation parameters defined in the configuration file. In this case study, some of the sensors are air density sensors. The definition of one of them is presented in Figure 2. We first choose the sensor name, specify the measurements type, data unit, how they are generated (randomly in this case study), the measurements interval –if needed–, the sending frequency and the queue to which data will be sent. We then define filters. In this case, we used a low pass filter and a high

[5] `https://github.com/kenzarh/IoT-Simulator/blob/main/simulation_config_casestudy1.json`

Table 1. First case study simulation parameters.

Sensor Type		Air density	Humidity	Pressure
Number of sensors		15	15	20
Seconds between two messages		10	10	10
Filters	Low-pass filter	1300 g/m3	-	1000 mbar
	High-pass filter	1000 g/m3	-	900 mbar
Perturbations	Data alteration	Random: [5,20]	Random: [5,20]	-
	Data deletion	-	-	Random: [3,15]
	Temporary stops	Each 8 seconds, Stop duration: 2 seconds	Each 8 seconds, Stop duration: 2 seconds	-
Final stop	Elapsed time	2 minutes	-	-
	Sent messages	-	60 messages	-

pass filter to bound the values between 1000 and 1300. The perturbations we inject are an alteration of data each 5 to 20 seconds, this frequency is chosen randomly between these two values, and a second perturbation that pauses the sensor each 8 seconds for 2 seconds. The final stop condition in this example is the sensor operating time that we limit to 120 seconds. Each sensor of the 50 used in this case study are defined the same way in the configuration file.
The results of the simulation, taken from the CSV file that contains metrics are presented as follows:

- Simulation duration: 6min 25s
- Number of sent messages: 1000
- Message sending frequency: 2.59
- Number of deleted messages: 459
- Message deletion frequency: 1.19
- Number of altered messages: 413
- Message alteration frequency: 1.07
- Number of stops: 283
- Stops frequency: 0.73
- Total duration of stops: 9min26s
- Number of filtered values: 262

This case study shows that the simulation respects the configuration file: it stops when 1000 messages are sent, filters and perturbations are used throughout the simulation.
It would be interesting to clarify the value of the metric “total duration of stops” which is greater than the simulation duration. The metric sums up stop durations from all the sensors in the simulation, we can have multiple sensors stopping at the same time which explains the value of 9min26s.

Fig. 2. Definition of a sensor and its operating parameters in the configuration file.

4.3 Second case study

In this second case study, we present a "smart" virtual sensor that uses a machine learning model to predict temperature using measurements of air density, humidity and pressure received from the RabbitMQ broker. We create a new class `TemperaturePredictor` that inherits from the abstract class `EditorConsumerSensor`. It enables the instantiation of a virtual sensor that retrieves air density, humidity and pressure values from the appropriate queues and predicts the temperature value on the basis of the three measurements using a pre-trained machine learning model.

The database The data comes from a Kaggle competition[6]. It consists of meteorological data measured in the German city of Jena every 10 minutes for 8 years, resulting in a total of 420 551 measurements. The

[6] `https://www.kaggle.com/pankrzysiu/weather-archive-jena`

Table 2. Output metrics for the sensors - second case study.

Sensor type	Number of sent messages	Sensor operating time	Number of filtered values	Number of deleted messages	Number of altered messages	Number of stops	Stops duration in seconds
Air density	150	30min 4s	127	150	150	149	298
Humidity	192	30min 1s	0	0	116	126	252
Pressure	357	30min1s	81	147	0	0	0
Temperature	150	30min4s	0	0	0	0	0

database consists of 15 columns, only 4 were used for our study : the temperature in degrees Celsius, air density, humidity and pressure. The goal of this selection is to simplify the study, since model evaluation is not central for this application.

Metrics used to evaluate the model For the evaluation of the machine learning model, we used some metrics that suit regression problems since temperature is a continuous measure.

- Root Mean Square Error (RMSE) is the most widely used metric for regression tasks. This index provides an indication of the dispersion or variability of the prediction quality. It tells how concentrated data is around the line of best fit. Its formula is:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

  where N is the sample size, y_i the actual value and $\hat{y}_i$ the predicted value using the model.
- Mean Absolute Error (MAE) is the sum of the absolute value of errors divided by the number of data points.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

The machine learning model After testing the different regression algorithms on our dataset, the best results are obtained with the Nearest Neighbours (NN) algorithm. We used the scikit-learn predefined algorithm "KNeighborsRegressor" set to 6 neighbours. The choice of 6 neighbours was done after testing different numbers of neighbours. Figure 3 represents the root mean squared errors for this model using 2 to 15 neighbours. The lowest RMSE is attained with 6 neighbours.
This sensor is integrated in our simulator to show that it is extensible and it can use any kind of sensors depending on the user's needs.

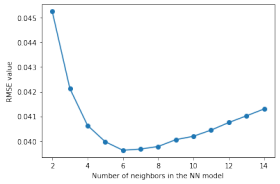

Fig. 3. RMSE using 2 to 15 neighbours

The simulation [7] We define four sensor instances: three sensors that send random values of air density, humidity and pressure respectively to the RabbitMQ broker and a fourth sensor, an instance of a smart sensor with the `TemperaturePredictor` class.
Table 2 presents the metrics for each of the four sensors at the end of the simulation .
This case study shows that our platform can integrate any kind of sensor, the extensible code facilitates this integration and enables to try different scenarios depending on the context and the user's needs.

5 Conclusion

In this paper we introduced a platform for simulating IoT exchanges. We presented it as a generic model that can be easily extended. It enables to define the sensors to use in the simulation and all their characteristics, filters, perturbations to be injected and stop conditions. We built a virtual sensor that predicts temperature using other sensors' measurements and based on a pre-trained machine learning model. This shows the extensibility of the model and its ability to introduce any kind of sensor in the platform. At the end of the simulation, metrics are returned enabling the evaluation of the simulation.
The remaining component of our platform is the real time visualization of the exchanged data so that the user can easily supervise the simulation. The long-term perspectives of this work are to be able to supervise IoT systems [X]We plan to define realistic scenarios of running with and without events. These scenarios will be shared and will allow us to compare different algorithms for decision-making, state monitoring and system resilience.

[7] `https://github.com/disp-lab/IoT-Simulator/blob/main/simulation_config_casestudy2.json`

Bibliography

[1] Bakare B, Enoch J (2019) A Review of Simulation Techniques for Some Wireless Communication System. International Journal of Electronics Communication and Computer Engineering 10(2):60–70

[2] Chernyshev M, Baig Z, Bello O, Zeadally S (2017) Internet of Things (IOT): Research, Simulators, and Testbeds. IEEE Internet of Things Journal 5(3):1637–1647

[3] Downard IT (2004) Simulating Sensor Networks in ns-2. Tech. rep., NAVAL RESEARCH LAB WASHINGTON DC

[4] Giménez P, Molína B, Palau CE, Esteve M (2013) SWE Simulation and Testing for the IoT. In: 2013 IEEE International Conference on Systems, Man, and Cybernetics, IEEE, pp 356–361

[5] Mallanda C, Suri A, Kunchakarra V, Iyengar S, Kannan R, Durresi A, Sastry S (2005) Simulating Wireless Sensor Networks with OMNeT++. submitted to IEEE Computer

[6] Naoumov V, Gross T (2003) Simulation of Large Ad Hoc Networks. In: Proceedings of the 6th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems, pp 50–57

[7] Park S, Savvides A, Srivastava MB (2000) SensorSim: A simulation Framework for Sensor Networks. In: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems, pp 104–111

[8] Saidallah M, Fergougui A, Elalaoui AE (2017) A Survey and Comparative Study of Open-Source Wireless Sensor Network Simulators. International Journal of Advanced Research in Computer Science 7(3)

[9] Singh CP, Vyas O, Tiwari MK (2008) A Survey of Simulation in Sensor Networks. In: 2008 International Conference on Computational Intelligence for Modelling Control & Automation, IEEE, pp 867–872

[10] Sobeih A, Hou JC (2003) A simulation framework for sensor networks in J-Sim. University of Illionis at Urbana-Champaign UIUCDCS

[11] Varga A (2019) A Practical Introduction to the OMNeT++ Simulation Framework. In: Recent Advances in Network Simulation, Springer, pp 3–51

[12] Varga A, Hornig R (2008) An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, pp 1–10