Master's Thesis

IoT WORKLOAD GENERATOR

Andreas Ioannou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

December 2020

IoT WORKLOAD GENERATOR

Andreas Ioannou

This thesis

was submitted in partial fulfilment of the

requirements for the Degree of Master of Science

at the University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

December, 2020

APPROVAL PAGE

Master of Science Thesis

IoT WORKLOAD GENERATOR

Presented by

Andreas Ioannou

Research Supervisor ____

George Pallis

Committee Member ____

Marios Dikaiakos

Committee Member ____

Dimitris Trihinas

University of Cyprus

December, 2020

Acknowledges

I would like to thank my family for the support they gave me during the whole years of my studies. I am also grateful to my professor George Pallis that introduced me to this topic and gave me this opportunity and specially to my supervisor Moysis Symeonidis, for their guidance and their support whenever I needed help during this difficult final year of my studies, which helped me complete this thesis and learn a lot of new things.

Abstract

Nowadays, the novel IoT application era has risen to the top, reaching a point that smart devices can also communicate between them. This new era introduced the need of continuous information exchange between the smart devices, which has as result that IoT applications produce vast amounts of data that need processing, and some of the applications require this processing to be of low or ideally no latency. The Fog and edge computing paradigms had emerged from these demands, as a solution, bridging the gap between users' applications and the cloud by providing intermediate processing nodes, closer to the user application, that minimizes the processing latency, which is essential for time-critical applications.

Deployment of fog systems can be really complicated, and it demands effort, time and costs, making the life difficult for users that would like to test or evaluate their IoT applications. The solution for this challenge is the paradigm of fog emulators, along with workload generator softwares. With an IoT workload generator integrated with a fog emulator, users can produce or replay sensor data based on their sensors' model description, and evaluate their applications with real IoT devices scenarios, by emulating real sensors easily, and without the need to purchase and set the real infrastructure.

This thesis introduces a novel IoT workload generator that (re)produces data from IoT devices in order for the users to be able to stress, evaluate and test their applications easily. Users can provide sensor models information, or recorded sensor data through datasets and the system will replay or produce new sensor messages data stream straight to the user's application. The implementation provides a comprehensive sensor model realization, along with a scalable, extendable and heterogeneous workload generator that gives the ability to the users of IoT applications to produce IoT data stream to their applications based on the user's input sensors configurations, regardless of the output data protocol or message format. On top of that, users can provide datasets of recorded real data that can be reproduced, with the ability to be customized, regarding the user's preferences, and finally they can implement the existing input and output interfaces for new features matching their specific needs.

Table of Contents

Chapter 1: Introduction 1

Chapter 2: Background Knowledge and Related Work........................... 5

2.1 Background Knowledge 5

2.1.1 IoT 5
2.1.2 IoT messages protocols.. 6
2.1.2.1 Request-Response Protocols.. 7
2.1.2.1.1 HTTP 7
2.1.2.2 Publish/Subscribe Protocols... 7
2.1.2.2.1 MQTT 8
2.1.2.2.2 KAFKA... 9
2.1.3 Cloud - Fog - Edge Computing 9
2.1.4 Fog emulators..11
2.1.5 Workload Generators...12

2.2 Related Work 13

Chapter 3: System Description 17

3.1 System requirements 17

3.1.1 Heterogeneity 17
3.1.2 Sensor Model realization 18
3.1.3 Accuracy..19
3.1.4 Extensibility - Configurability- Customization...........................19

3.2 System overview..20

3.2.1 System Capabilities and functionalities 20
3.2.2 High Level System Architecture 22

Chapter 4: Model Mapping... 26

4.1 Modelling of IoT Sensors...28

4.1.1 Input file, Sensor Prototypes & messages 29
4.1.2 Produce Mock data feature modelling......................................32
4.1.3 Replay data from dataset feature modelling.............................35

4.2 Output Mapping 39

Chapter 5: Implementation details 41

5.1 Main Users actions..42

5.1.1 Start of Workload Generator 43

5.1.2 Stop of Workload Generator 46

5.1.3 Restart of Workload Generator 46

5.2 Extendable Interfaces Overview...47

5.2.1 Output Protocol Interface..48

5.2.2 SensorPrototype Input Interface 50

5.3 Produce Mock Sensor Data Implementation 53

5.3.1 Components and Association 53

5.3.2 Class Modelling 55

5.3.3 Flow...57

5.4 Replay sensor data from Dataset Implementation...63

5.4.1 Components and Association 63

5.4.2 Dataset Thread Modelling..64

5.4.3 Flow...65

5.5 Output protocol Implementations..68

5.6 API Endpoints 70

5.6.1 Workload Generator Actions APIs 70

5.6.2 Sensor Actions APIs..72

Chapter 6: System Evaluation and Results... 76

6.1 Usability Evaluation...77

6.1.1 Heterogeneity and Sensor Model Realization Evaluation..........................77

6.1.2 Accuracy Evaluation 82

6.1.3 Extensibility, Configurability, Customization Evaluation 85

6.1.4 Usability evaluation, through Integration with real Fog Emulator on a real scenario use case...85

6.1.4.1 Increasing Workload Test...90

6.1.4.2 Data Skewness Experiment 92

6.2 Performance Evaluation and limitations 94

6.2.1 Sensor Message Delay..95

6.2.1.1 Input and Evaluation Methodology 96

6.2.1.2 Evaluation Experiments...100

6.2.1.2.1 Sensor Quantity Factor...100

6.2.1.2.2 Generation Rate Factor 104
6.2.1.2.3 Host Machine Resources Factor 107
6.2.1.2.4 Sensors Quantity Limitation 109
6.3 Evaluation Results Summary and System limitations 112

Chapter 7: Conclusion and future work 115

7.1 Synopsis 115

7.2 Conclusion 116

7.3 Future work 117

Bibliography and References 118

Appendix A 121

A.1 Repository of Source Code for implementation 121

List of tables

Table 6.2.1.2.1 1: Experiment results for input sensors number factor 101
Table 6.2.1.2.3 1: Experiment results for hosted machine cores factor 107
Table 6.2.1.2.4 1: Input Sensors number Performance Limitation on a machine with 2 available cores... 110
Table 6.2.1.2.4 2: Input Sensors number Performance Limitation on a machine with 8 available cores... 111
Table 6.2.1.2.4 3: Input Sensors number Performance Limitation on a machine with 16 available cores... 111

List Of figures

Figure 3.2.2 1: System High Level Architecture 25
Figure 4.1.1 1: SensorMessagePrototype Structure 30
Figure 4.1.1 2: SensorPrototype Model Diagram 31
Figure 4.1.2 1: MockSensorPrototype Input Example 33
Figure 4.1.2 2: Sample output for the input in 34
Figure 4.1.3 1: DatasetSensorPrototype Input Example........ 37
Figure 4.1.3 2: Motion_detection_output.csv instance 37
Figure 4.2 1 :Output Protocol Configs........ 40
Figure 5.1 1: State Transition diagram for Workload Generator........ 43
Figure 5.1.1 1: Flow Diagram for **Start** User Action 45
Figure 5.2 1: Exchange Date Model 47
Figure 5.2.1 1: ISensorMessageSendService Interface along with its current implementations class model diagram 48
Figure 5.2.2 1: ISensorDataProducerService Interface with its current implementations diagram 51
Figure 5.3.1 1: Produce Mock Sensor Data Flow 54
Figure 5.3.2 1: Produce Mock Sensor Data class relation model diagram 56
Figure 5.3.3 1: Produce Mock sensor data flow diagram 59
Figure 5.3.3 2: Scenario Management Class Diagram. 61
Figure 5.4.1 1: Replay Sensor Data from Dataset Flow........ 64
Figure 5.4.2 1: Replay Sensor Data from Dataset Main Thread Class Diagram 65
Figure 5.4.3 1: Replay Sensor Data from Dataset Thread flow diagram........ 67
Figure 6.1.1.1: Screenshot from a dataset that measured annual photovoltaic panels data in Cyprus 78
Figure 6.1.1.2: Sensor Modelling Input inside configs_file.json for dataset from Figure 6.1.1.1 78
Figure 6.1.1.3: Screenshot of a dataset that measured beach water quality in Chicago 79
Figure 6.1.1.4:Sensor Modelling Input inside configs_file.json for dataset from Figure 6.1.1.3 79
Figure 6.1.1.5: Screenshot of a Yellow Taxi Trip Data from NYC dataset 79
Figure 6.1.1.6: Sensor Modelling Input inside configs_file.json for dataset from Figure 6.1.1.5 80
Figure 6.1.1. 7: Road traffic data instance from a dataset for a city In Denmark 80
Figure 6.1.1.8: Sensor Modelling Input inside configs_file.json for dataset from........ 81
Figure 6.1.2 1: Input file that illustrated the data for the sensor message accuracy validation 83
Figure 6.1.2 2: Exported first rows data from the temperature_output.csv file exported by the workload generator for the data accuracy validation experiment based on input in 84
Figure 6.1.4 1: Real topology of scenario........ 87
Figure 6.1.4 2: Emulation topology using Fogify on a virtual machine for the experiments... 88
Figure 6.1.4.1 2: Base Input file of workload generator for scenario........ 89
Figure 6.1.4.1 1: StreamSight Query for the average temperature of 5 minutes........ 90

Figure 6.1.4.1 3: Process latency based on the sensor number 91
Figure 6.1.4.1 4: Latency of processing for each batch based on the sensors number(sensor messages number) 92
Figure 6.1.4.2 1: StreamSight Query for the average temperature of 5 minutes, grouped by brightness.. 93
Figure 6.1.4.2 2: Results of experiment for skewed workload 93
Figure 6.2.1.1 1: Base Input file for sensor message delay accuracy experiments................ 97
Equation 6.2.1.1 1 : Equation used in order to calculate the average sensor message delay of a MockSensorPrototype 99
Figure 6.2.1.2.1 1: Graphical representation based on the results in Table 6.2.1.2.1 1 for the sensor number factor experiment 101
Figure 6.2.1.2.1 2: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 1000 103
Figure 6.2.1.2.1 3: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 2000 103
Figure 6.2.1.2.1 4: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 3000 103
Table 6.2.1.2.2 1: Experiment results for generation rate factor 105
Figure 6.2.1.2.2 1: Graphical representation based on the results in Table 6.2.1.2.2 1 for generation rate = 6 s 105
Figure 6.2.1.2.2 2: Graphical representation based on the results in Table 6.2.1.2.2 1 for generation rate = 10 s 106
Figure 6.2.1.2.3 1: Graphical representation based on the results in Table 6.2.1.2.3 1 for generation rate = 2 s in machine with 16 cores.. 108
Figure 6.2.1.2.3 2 : Graphical representation based on the results in Table 6.2.1.2.3 1 for generation rate = 6 s in machine with 16 cores.. 108
Figure 6.2.1.2.3 3: Graphical representation based on the results in Table 6.2.1.2.3 1 for generation rate = 10 s in machine with 16 cores.. 108

Acronyms

IoT	Internet of Things
HTTP	Hyper Text Transfer Protocol
MQTT	Message Queuing Telemetry Transport
AMQT	Advance Messaging Queue Protocol
CoAP	Constrained Application Protocol
OS	Operation System
JSON	JavaScript Object Notation
REST	REpresentational State Transfer
URI	Uniform Resource Identifier
API	Application Programming Interface
QoS	Quality of Service
M2M	Machine to Machine
IP	Internet Protocol
TCP	Transmission Control Protocol
HTML	HyperText Markup Language
CSV	Comma-Separated Values

Chapter 1

Introduction

IoT devices transform the physical world into a cyber-physical dynamic environment. Devices can now connect, interact, collaborate and communicate not only with humans, but even between them. The later ability creates a new potential to bring immense value into our lives. Thus, the bidirectional exchange of data between devices and humans helps us to amplify our abilities, and, consequently, makes our everyday life easier and more convenient. To this end, a novel IoT application era is created that boosts multiple sectors such as health care, industrial automation, transportation, smart-cities and of course our daily personal convenience. Just to get an idea of the potential use cases of IoT in our daily life, imagine a smart house that can monitor itself on its own without any human intervention. This smart house, can consist of smart lambs that can turn off if they are not needed, sensors that can sense fire or something not normal in the house, or cameras that monitor any strange behaviour and inform the owner, providing more security and safety to our life, and on top of that, saving money and energy for a better environment. However, all these IoT devices and applications produce much more data than their early stages while they are getting smarter. Recent calculations [1] estimate that every second 127 new devices get connected to the internet, which results in the estimation that 75 billion IoT devices will be online by 2025. The later contributes to a parallel increase of the generated IoT data. For instance, according to a new IDC forecast [2 , 3], connected IoT devices are expected to generate 79.4ZB of data in 2025 with more than 59 zettabytes (ZB) of data to be created, captured, copied, and consumed in the world this year.

This increased number of devices comes with an increased heterogeneity in terms of technologies, data types, formats and protocols. The combination of the heterogeneity and the vast amount of streaming data is one of the main bottlenecks in application development, testing and evaluation. In addition, with this growth of interconnected devices, which implies growth of generated data, the produced data volume rates cannot be controlled in a real testbed. Furthermore, the cost and the effort needed in setting and deploying hundreds or thousands of IoT sensors are only a few of the difficulties that users need to face and address for their new IoT systems to be tested. For a better understanding, let us introduce a representative use-case of a smart-city application that facilitates in monitoring of the city traffic and notify people for possible accidents. In such IoT service, distance between sensors, devices, servers etc, could be tens or hundreds of kilometres with the operator to be forced to waste a lot of time in order to just set a deployment for testing. Not only that, for a single change, the operator should redeploy and readjust partially (or even totally) the system. It is reasonable for this long back-and-forth procedure, to be costly in money for renting or purchasing the devices, and in terms of time and effort for placing IoT in their physical locations. In order for the users to bypass the above problems, they are forced to spend time in writing software-based IoT data generators of diverse devices, or waste time to find commercial software suites, where unluckily for them, they are not focusing on IoT data, hoping that they can find the right software to satisfy each user's needs. Having said that, the question here, is whether a system can be implemented, and how, in order to address all the above-mentioned challenges for IoT applications in fog computing.

The current thesis answers the above question. In order to tackle all the above issues, we introduce an all-in-one IoT data generation framework, where the user provides a model representation of her sensors' properties and the system creates thousands of sensors in order to produce the data that the user's application could produce in real life, making easier the integration and testing of real scenarios and handling the heterogeneity of the sensors nature and properties at the same time. In addition, our work can also replay real data from CSV datasets, with the ability to customize them, providing to the user the choice to use the same data from the dataset or mix real data with users' preferences and customizations to produce synthetic data. Another great feature of our framework is that it supports the heterogeneity found in protocols and data representation of IoT devices. Generated data can be represented as JSON, XML, TXT etc. and can be published through HTTP, MQTT or using KAFKA broker. On top of that, users can extend the functionalities through programmable Interfaces in order to create for example more output protocols or add more input configuration features. Last but not least, statistical analysis of the data populated, dynamic change of the data and export of the generated data to dataset files, in order to be analysed or replayed later on if they want, are also supported.

Based on the above supported features the contributions for this work are: 1) A comprehensive model for describing IoT data generation, 2) a scalable multithreaded open source framework that simulates as closer to the reality the desired model 3) an heterogeneous IoT workload generator 4) a system that support experimentation with real-world scenarios and evaluation of their performance individually, or integrated with fog or edge emulators.

The rest of the thesis is structured as follows: In Chapter 2 a basic background knowledge with some terms and technologies will be introduced in order to get familiar with the context, along with some related works, while in the

Chapter 3 the first introduction of the system will be stated, with the requirements and an overview of the architecture. Chapter 4 presents some basic model mapping between the input and the entities of the system, but more details and implementation-wise information can be found in Chapter 5. Experiments and results for the system evaluation are explained in Chapter 6 and in the end, Chapter 7 concludes the thesis.

Chapter 2

Background Knowledge and Related Work

2.1 Background Knowledge
- 2.1.1 IoT
- 2.1.2 IoT messages protocols
- 2.1.3 Cloud - Fog - Edge Computing
- 2.1.4 Fog emulators
- 2.1.5 Workload Generators

2.2 Related Work

2.1 Background Knowledge

2.1.1 IoT

Recently, a new era of applications was formed when devices started having abilities to do more intelligent stuff, exceeding their normal tasks. Devices can now communicate, learn [4], advise or even act on their own. In order to do more smart actions, these “smart” devices need to observe, sense and form the state of the environment they are located at any moment, using multiple sensors, or by exchanging information with other devices, in a way similar to what we as humans do, using our senses. These new capabilities for the devices were enabled from advancements in microelectronics and low-power communication protocols, improvements in wireless sensor networks, machine learning techniques and other technologies. All these improvements or new techniques gave birth to this novel era that we call IoT so that it can drive “intelligence” in interconnected smart devices or portable objects (aka “things”) [5 , 6 , 7].

The IoT era, brought potential and benefits to numerous sectors of our daily life, and lifestyle. Health care, industrial automation, transportation, resource monitoring and management are just a fraction of the benefited sections by IoT. Starting with health care, mobile tools for telemedicine, remote patient tracking technologies via wearables, or even unmanned aerial vehicles that can provide help, in conditions that human lives could be in risk, are just a few examples. Furthermore, smart houses using monitoring sensors, can monitor themselves in order to reduce energy consumption, and provide security and safety to the owners. In addition, smart cities using smart traffic lights and optimized public energy grids can offer traffic monitoring, improvement of resources consumption and efficiency. In transportation, interconnected autonomous cars, can drive themselves without human intervention, using cameras, radars, and other sensors to capture information about roads, traffic, and to prevent potential accidents. All of the above examples will result in devices that can be self-controlled, and humans will be released from the stress to monitor them and the effort do the tasks on their own.

2.1.2 IoT messages protocols

A challenge for the IoT application developers, is to decide which output protocol their sensors message will have. The reason for this is that there is no official standardization yet for IoT devices protocol. Hence multiple data transfer protocols can be used, for example HTTP, MQTT, AMQT, CoAP etc. each with its advantages and drawbacks [7 , 8].

2.1.2.1 Request-Response Protocols

2.1.2.1.1 HTTP

HTTP [9] is the most widely used and known data transfer protocol due to the internet explosion. HTTP protocol is a request–response application layer protocol in client-server computing model. Clients (e.g.: a web-browser) send request to the server, and the server replies with the requested resource in the response. For this type of protocol, the user must define the request URI endpoint that the HTTP request will be send to (the endpoint that the server will be listening to), and the clients or servers' information (host/IP and port) that has the requested resource. If web services of HTTP protocol conform the REST software architecture style, which implies that they are using a uniform and predefined set of stateless operations or generally follow some global good practises and conventions and a set of constraints that ensure a scalable, fault-tolerant and easily extensible system, they also can be considered RESTFull web services.[10]

2.1.2.2 Publish/Subscribe Protocols

Publish/Subscribe protocols differ from request-response in some basic concepts. For the publish-subscribe message pattern, we have 2 client entities, publishers and subscribers. In this design, devices or clients (called publishers) send-publish messages tagged with a topic. On the other end, other clients called subscribers, subscribe to topics, and they care about only the messages that are tagged with the topics they subscribed. Optionally and most of the times, in between there is an intermediate node, called broker server, that collects the messages from the publishers, group them based on the topics, and forwards them to the subscriber clients that had subscribed previously to the broker based on the topic. In case that the broker exists, neither the publishers know which subscriber will collect their messages, neither the subscribers know from which publisher the data came from. In the other case, that the optional broker server is not supported, publishers multicast the messages. The concept overall is that publishers send data using topics, and these data eventually will be collected from subscribers.

2.1.2.2.1 MQTT

MQTT [11] is a lightweight message queuing and transport protocol. It is a publish / subscribe network protocol with purpose the transfer of messages between devices. As its name implies, it is suited for transport of telemetry data (sensors and actuators) and along with its lightweight nature, it is suited for M2M communication and ultimately for the IoT. MQTT protocol can consists of 2 network entities, MQTT broker and clients (publishers or subscribers) .The publisher client sends messages to the MQTT broker tagged with a topic, and the MQTT broker distributes the messages to clients that subscribed to the topic. MQTT relies on the TCP protocol for data transmission and can support QoS. QoS is a measurement of the agreement between a sender of a message and the receiver that guarantees the delivery of the message. There are 3 QoS levels in MQTT, 1) at most once (sender sends the messages and does not expect acknowledgement of delivery of the message from broker or other clients) , 2) at least once (the message is being re-tried until acknowledgement is received) , 3) exactly one (sender and receiver establish two level-handshake in order to assure that only one copy of the message was received). The configurations needed for this output protocol are broker servers' information (ip and port) that the data will be published to, and the topic that they will be published to.

2.1.2.2.2 KAFKA

KAFKA [12 , 13] is not an output protocol, but more accurately an open source streaming platform currently supported by the Apache Foundation. It uses the publish-subscribe message pattern and has similarities and same concepts as MQTT. KAFKA's biggest different from MQTT protocol, is the scalability provided through the distributed and persistent data preservation, since KAFKA was designed to handle huge amount of streaming data. KAFKA also depends on Zookeeper. Zookeeper is a top-level software developed by Apache, and keeps track of the status of KAFKA cluster nodes, partitions, topic subscriptions etc. Asides from their differences, they share plenty of the information required in order to achieve data transfer. Users need to provide KAFKA broker servers information (port and ip) as well topics for the messages.

2.1.3 Cloud - Fog - Edge Computing

In order for devices to intercommunicate, and learn, they need data that is previously generated and collected, and can be accessed from everywhere. All these data produced are being stored in the *cloud*. Cloud or cloud computing [14] is the availability of computer system resources (computing power and data storage), at any time, in which the user does not have direct management access . All the generated data can be stored using cloud computing, on datacentres all over the world, and can be accessible through the Internet at any time. The distance between a machine that wants to retrieve some data, and the cloud server that has these data, can vary, in worst case, it could be some continents away. Nowadays, a lot of applications are time critical that cannot afford the latency created by the distance of the machine that stores or is needed to process the data in order to return the desired information. This latency can downgrade the performance for these time critical applications or even make them useless, if they need to process data in real time (some connected/driverless transport applications for example, where split-second reactions may be essential). On top of that, IoT devices export huge amounts of raw data that they record every second, from which only a small fraction can be useful, increasing the data traffic around the internet, creating network congestion which will result to more latency to retrieve the desired information.

This increasing data rate and real-time analysis requirements create the need of a new approach to address them. From these requirements, *edge and fog computing* [15 , 16 , 17] emerged. The basic idea behind these new approaches is bring the processing as closer to the user device or application as possible, by placing some intermediate nodes between the cloud servers and the devices or even on the devices. With this technique, we aim to bridge this huge gap between end devices and the cloud servers, to minimize the latency and also handle properly the data stored to be only the valuable ones, to earn cloud storage.

Fog computing [15], is the method to place processing nodes, geographically in between the devices and the cloud servers. This new layer of processing, located between the devices and the cloud, consists of the fog nodes. In the lower level of the fog layer (closer to the devices), we can often find low-capacity machines that are located in the local network of the devices, and by moving from lower to higher level in the fog layer, the capabilities and the processing power of the fog nodes is being increased, where in the top level we can find the cloud, that has theoretically unlimited processing resources and capabilities. The purpose of this newly created layer, is to process as much as possible the incoming data stream produced by the devices, with the goal to handle the problem of latency in processing for time-critical services, such as an autonomous car service that cannot afford processing in geographically-far nodes, and on top of that with this “local” processing, is being implied that the amount of data sent to the cloud is being reduced, cutting out unnecessary data transfers, simplifies cybersecurity, decreases network and system response times and network congestion.

An even more enhancement of the fog computing approach is edge computing, which is the movement of the processing on the devices, eliminating even more the latency from applications to the fog nodes. This processing that can happen on the end users devices, would be the best solution, but usually edge nodes lack capacity and processing power in comparison to fog nodes, in order to make them more usable, practical and feasible (a mobile phone or sensor with more resources usually implies bigger size and to be more expensive).

This rising combination of edge and fog computing will support users' needs for greater performance and at the same time reduce the big data problem that it is meant to grow within this new IoT era.

2.1.4 Fog emulators

Fog and edge computing brought some challenges to for users that want to develop, deploy and test their IoT applications. In order for operators to just evaluate a single scenario of their IoT application in fog, they need to purchase, place, program and deploy the sensors, where this long procedure costs them in terms of money, valuable time and effort. In addition, this whole procedure needs to be reproduced for maybe each different testing scenario. These challenges for fog operators can be addressed by emulating the application instead of real deployment of testbeds.

Emulators are hardware or software tools that can realize or represent the behaviour of an application or a system at a host device or host infrastructure by minimizing the real deployment cost. These systems are useful due to the ease that the user can manipulate the input or the state of a runtime application. Their main functionality is to represent the behaviour of the application, before it can be used in the real context, with users achieving less time and cost waste, as well as effort.

Fog-computing emulators need to be able to provide an execution environment that captures realistically the conditions and behaviour of a Fog application deployment, focusing on resource heterogeneity, controllable failure scenarios, scalable experimentation and monitoring capabilities. Fogify [18], is an example of a framework for fog-applications emulation, that aims in emulation of fog applications with the minimum effort for the users. Beyond the ease of the set up for the emulation, Fogify also supports runtime monitoring, assessment of deployment, resource and link heterogeneity and scheduled scenarios that can provide run-time changes for more wide test cases. Using the containerization infrastructure of both this work, and Fogify framework, the workload generator implemented, can be used perfectly along with Fogify, in order to provide a complete fog computing emulator, than can emulate and produce IoT devices data, providing a powerful tool to any IoT fog application developer with the ability to deploy, experiment, inject faults or different scenarios' data stream and finally test their fog applications before introduced to public.

2.1.5 Workload Generators

Fog emulators cannot generate data, not without a workload generator. Workload generators are software tools with purpose to generate synthetic workload for a benchmark or a real application in order to emulate the behaviour of end users. Usually, the input to such softwares are some models or properties regarding the nature or the required data that will be generated. The workload to be generated can be data or even random calls just to stress an application and see how it would handle enormous amounts of requests that could happen in real life. There are a lot of types of workload generators, based on the application nature or the context of the application. Some main categories of workload generators are web-based systems' [19] workload generators, database workload generators, CPU stress workload generators etc. Maybe the biggest and most used category nowadays is the web-based workload generators, due to the explosion of the internet applications and trends. For the context of this thesis, an IoT workload generator for fog computing will be described.

2.2 Related Work

The need for data generation frameworks, capable of modelling and generating rich data distributions from real-world data, was an open issue even before 2000s [20]. The first attempts to solve this problem came from the data management community [20 , 21]. These solutions provide domain-specific languages for dataset descriptions and well-implemented systems for dataset generation. However, they lack in the generation of streaming data, and most of these solutions are not web-based.

While the computations are being migrated from Cloud towards network Edge, it makes the process of evaluating computing systems even more challenging [22] and on top of that, the lack of comprehensive datasets leads to a need for synthesizing various datasets [23 , 24]. Covering this need, will also identify verification and validation of software, optimization of algorithms, and augmenting existing data with synthetic, as the benefits of synthetic data generation in an industrial IoT scenario. A recent attempt at datasets synthesis, is proposed in [23], where authors create a synthetic dataset from trajectory data, hotspot locations, and Wikipedia traffic. Similarly, Spaten [25] combines data from social media, POIs, and configurable parameters to create large synthetic datasets. Interestingly in [26], authors utilize Hadoop in order to implement a scalable generator of IoT data. The distributed nature of the system guarantees fast and large-scale execution, while the generated datasets reach thousands of gigabytes in size. However, all the above solutions lack generality, generate static datasets not streaming data, and are not easily extensible.

On the contrary, ELIoT [27] is an emulated IoT platform that provides out-of-the-box specific IoT sensors, like weather sensors, and emulates them realistically. Furthermore, Resense [28] is a system capable of replaying captured data from real sensors. Specifically, Resense captures real data from sensors and, when the user needs to reproduce them, it emulates "fake" sensors on edge devices, like raspberries, re-producing the captured IoT data. Even if those systems provide streaming generation of the data, are not easily extensible or are tailored only to specific sensor types.

Beyond the publications and research papers, plenty of commercial and professional tools and projects exist in repositories or on the internet that can generate specific IoT data. Httperf [29], for example, is a general purpose widely used tool for measuring web server performance, which also provides an embedded workload generator that produces HTTP requests. It is used mostly to stress a server with HTTP heavy workload in order to see the performance under a heavy number of requests. On the other hand, If the user wants to use MQTT protocol for the output messages, he can use for example the *aws-iot-mqtt-load-generator* [1] project that works ideally for IoT devices and MQTT output protocol. This system supports some

[1] https://github.com/amazon-archives/aws-iot-mqtt-load-generator

nice features, like Random generation functions that can be used for numbers by giving min and max values or generating text by using input weights to manipulate the random text generation or even expressions functions to generate the data. Furthermore, Apache JMeter [2], is an open source application designed to measure performance initially for web applications, but it has been expanded to support other test functions. One of the key features of JMeter, is the ability to process and evaluate HTTP calls, simulating the users submitting request to servers, and can measure the performance of the servers response times along with other metrics provided in user created tests. It can be extended, and configured and in addition can extract and process data from different response formats such as JSON, HTML, XML, etc . On top of that, it can send HTTP requests including payloads declared by the user and can also send requests with other output protocols using plugins. Even the tool provides numerous important features, and fulfils more than enough requirements and needs of a lot of web-based users, it cannot be easily used for generation for sensor-wise data stream, based on users sensor models. This is the most important reason that is not ideal for our context, since the support of a realization model that can comprehend the sensors, along with the generated data, based on user's model configurations, is mandatory for IoT application evaluation on emulators. All the above systems are decent, and can cover some or all of the needs of some users, but are not ideal for IoT applications or lack generality, which implies that many other IoT application users will not be benefited for emulation of their applications on fog or generally. On top of that, most of them are not extensible, and only few support multiple output protocols. Having said that, the wide range of protocols that IoT devices can use to exchange data (AMQP, CoAP, MQTT, HTTP etc.) [7 , 8], enhanced by the non-standardization for IoT data protocols and the wide area of context for IoT application use cases, are creating extra challenges for the

[2] https://jmeter.apache.org/

users to find the ideal workload generator, especially if their system supports multiple IoT data protocols, or different specifications. The difference between all the existing work, and this implementation, is the support of the Heterogeneity existing under a lot of pillars in this novel IoT era (one of the most important is the support of multiple IoT output protocols), the scalability provided in terms of load generation, the ease of integration with fog computing emulators or simulator systems, or even individually and finally, the combination of a comprehensive sensor realization model with the provision of a general data stream generation framework that can be used or easily extended from different users or operators to cover all their needs, regardless of the use case or application specifications under the IoT context.

Chapter 3

System Description

3.1 System requirements

3.1.1 Heterogeneity

3.1.2 Sensor Model realization

3.1.3 Accuracy

3.1.4 Extensibility - Configurability- Customization

3.2 System overview

3.2.1 System Capabilities and functionalities

3.2.2 High Level System Architecture

In this chapter, the requirements that an IoT workload generator should have will be described, as well as an overview of the implemented functionalities and features of the system will be introduced.

3.1 System requirements

3.1.1 Heterogeneity

The rise of the era of IoT, brought some challenges for users that want to deploy their IoT applications in a fog environment. The amount of different contexts of applications, the lack of standardization in IoT protocols, and the wide range of different sensor properties, amplifies the support for heterogeneity in different levels, for IoT data generators. Users should be able to use a workload generator without restriction to their application context or their sensors’ nature. On top of that, a decent workload generator should be able to support multiple data representations, or formats and ideally to be easily extensible for the developers to include more in the future. A normal fog application could consist of multiple different IoT devices and sensors, that each one could communicate with different output protocols. This common scenario should not force the users to use different workload generators for each sensor protocol, neither to limit their application’s universal testing into divided test cases.

3.1.2 Sensor Model realization

One of the main requirements of an IoT workload generator is to be able to represent sensors as close as possible to reality. In order for this to be achieved, an accurate and representative realization model should be supported. Users firstly, need to extract the essential properties of their future emulated sensors from statistics of already exported real sensors data, or at least provide an estimation of the properties for their desired sensors. This should be the only action needed from the users, since after extracting and acquiring these properties and configurations, they should be able to easily provide these properties of the sensors in an abstract model that describes accurately or with approximation the real sensors, and the system to comprehend and realize the future created and emulated sensors, as real sensors that export real data. An ideal model realization should be able to support not only the common sensor properties shared by multiple different sensor types, but on top of that to support also the unique tailored properties or configurations for any IoT sensor. This would provide the ability to any IoT application user, to describe his IoT sensor’s essential attributes and properties, and provide them to the system that would eventually emulate the sensors accurately, and produce data as close to the real data produced by the real sensors.

3.1.3 Accuracy

One of the most essential requirements of an IoT workload generator is the accuracy of the data generated from the emulated sensors. The accuracy can be divided in 2 categories, the accuracy of the values of the generated data and the accuracy of the time that the data will be produced, or replayed. Having said that, firstly it is needless to point out the importance of the accuracy regarding the generated data stream from the workload generator. The workload generator should produce data that match and correspond to the configurations of the model provided, which implies that the emulated generated data could be data that the actual real sensors could produce. Beside the correct sensor emulation and accurate data production, workload generators should also be able to handle time-sensitive applications. The data generation rates in these kinds of applications should be retained with priority, in order for the users to test their application more correctly for real scenarios. On top of that, for the emulation of a timestamped dataset, it is redundant to point the importance for accurate message time population. In cases that users have the luxury to use exported datasets of real sensors, it is mandatory for the system to be able to replay the same exported data as they are being exported from the dataset.

3.1.4 Extensibility - Configurability- Customization

It is not a mandatory requirement, but a workload generator that supports extensible architecture or customizable configurations, is a more powerful tool, and of course more appealing to the users. Extensible interfaces that guide the user to expand functionalities or add new features, create the potential for users to be able to customize the workload generator to fulfil their more specific needs. Furthermore, it is more useful and user friendly that the developer doesn't need to change manually things for each test or run. A system that supports configurable properties provides a more dynamic behaviour that boosts flexibility.

3.2 System overview

In this section, a first introduction to the system will be happen, stating and briefly explaining the capabilities of it, along with the main implemented functionalities and features. Then, a high-level overview of the framework architecture will be described, with introduction to its main components.

3.2.1 System Capabilities and functionalities

The system is a data generation framework that is capable of 1) producing IoT data stream based on users' configurations as much general as possible, by supporting the heterogeneity found in different IoT devices in terms of data representation, output protocols and message formats, and 2) to reproduce real sensor data from CSV dataset embedded with user customization, in order for any user to be able to test his IoT application's behaviour and performance, before real public deployment, easily, with convenience and accuracy. More specifically, the system has two main implemented functionalities, 1) the ability to produce mock data by emulating IoT sensors, based on the user's provided sensor models and configurations, and 2) the ability to replay sensor data from provided CSV datasets.

For the first main functionality, the user provides through the input, properties and configurations about his real sensors (e.g.: type of the sensors, how often should each type of sensor be generating a new sensor message, the fields that each sensor message will be consist of, the format of the message etc.), along with execution information (the amount of desired sensor to be created, the output protocol of each generated sensor message etc), in order for the system to emulate the desired sensors, and start producing mock data. On top of that, the system can export the generated sensor data stream to TXT or CSV output files, along with statistics for later use. Another powerful tool for this main functionality is the ability to alter data population for specific sensors during runtime, with declared by the user scheduled actions, called scenarios. Using these scenarios, the system can schedule some events that can occur at a specific time in the future, after the system started producing data, and these scenarios can trigger the alternation of the values for the future generated data for specific sensors, in order that the desired sensors will produce different data for a period of time, or follow a different data population method (more on this later).

Regarding the second main implemented functionality, the system can reproduce sensor data from CSV dataset, in which dataset are stored the exported sensor messages from real sensors. Beyond the simple action to replay the data exactly like they exist in the dataset, the system supports some extra features for this implementation. The framework can ignore the timestamp of the dataset sensor data if the user provides a custom generation rate, and replay the data based on the custom generation rate. In addition, the framework can sort the dataset using external sorting[3] and finally it provides the ability to enhance the replayed sensor messages from the dataset, with runtime mock populated values for the fields of each message, using part of the implemented features of the first main functionality (produce mock data).

Beyond the two main functionalities, the framework supports some extra features. To begin with, It supports heterogeneity in data population, by generating data using different data representations (Double, String, Integer, Boolean, Object), different sensor message formats (JSON, TXT, XML), different output protocols (HTTP, MQTT, to KAFKA broker) and different data population methods (using constant values, random number generation between number range, following normal distribution or probabilistic distributions). Last but not least, the framework, provides 2 fully extendable interfaces, one for input (to add more main functionalities beyond the 2 already implemented - produce mock data, replay data form dataset) and one for output (to output to more output protocols, beyond the 3 already supported - HTTP, MQTT, to KAFKA broker), that any developer can extend in order to add even more features and fulfil his own specific needs.

Overall, the framework can provide generation of mock or synesthetic IoT sensor data, and replay real sensor data.

3.2.2 High Level System Architecture

In general, the system consists of some input functionalities, along with some output functionalities. In this section, we will see the hight level architecture of the system, and explain the flow, in high level.

[3] https://mvnrepository.com/artifact/com.google.code.externalsortinginjava/externalsortinginjava/0.4.5

Figure 3.2.2 *1* depicts a high-level overview of the system's architecture in a representative flow of the initialization and the start of generation of sensor data stream. The architecture can be divided in 3 different layers, the **Control** layer, the **Emulation** layer and the **Output** layer. The *Control Layer* is responsible to handle and verify the input of the user, and pass the processed input to the emulation layer. The *Emulation layer*, which consist of multiple emulation instances, each one generating data according to tis implementation, produces the data stream, and pass it to the output layer. Finally, the *Output layer*, will forward the generated data stream to the user nodes, through the desired output protocol.

More specifically, the deployment starts with the users creating and providing the needed input files to the system. The framework in order to successfully start producing data, requires some files. The first file that is mandatory, it's called ***configs_file.json***, in which the user has to provide information about the sensors, the output protocol and output nodes. Optionally, the user can provide a configuration file called ***workloadGenerator.cfg*** that stores key-value pair of runtime properties, and finally the datasets if he demands from the framework to replay data from them. Once the user has prepared the files and provided them, he can start the workload generator through the controller's API, and then the ***Controller*** parses the files and verifies their validity. More accurately, it evaluates if the configurations are syntactically correct, if all the mandatory information are provided and ensures that there are enough available underlying resources for the IoT devices emulation. If any inconsistency exists, the *Controller* returns an error message to the user and terminates the process. Otherwise, the *Controller* establishes connections with the output nodes from the input, that the generated data stream will be send to, and using the *Coordinator* service, instantiates the emulated IoT instances by invoking the related interface method for all the desired implementations through the *Class* *invoke*r and providing to each one the required parsed and processed configurations. After that, each emulator instance, using the required processed information provided by the Coordinator, it will do its internal validations and will start generating sensor messages, based on its implementation. With the interfaces on hand and hosted in every emulated IoT instance, each of the emulation instances is responsible for their own threads, creating one thread per sensor and monitoring and coordinating them, where a sensor thread simulates a sensor connected to the emulated IoT device. Furthermore, a sensor thread generates values based on descriptions provided by users or can just replay IoT data saved in CSV files. At the same time, the system collects statistics from the emulation layer and the generated sensors' data. Thus, all produced data from the emulation instances are collected, and users can perform a post-execution analysis so as useful insight statistics to be created and exported or even users have the option to "replay" the same experiment in the future. Lastly, every emulated instance, packages the sensors' data into messages and propagates them to the output layer, which will handle the transfer to the required output destination nodes. With the messages received, the output processor is responsible to invoke the appropriate output protocol interface implementation for each message based on the user's description and disseminates the message to the proper application service through the defined channel (pub/sub, HTTP, etc.). Finally, all the produced data streams will be populated to the user's application.

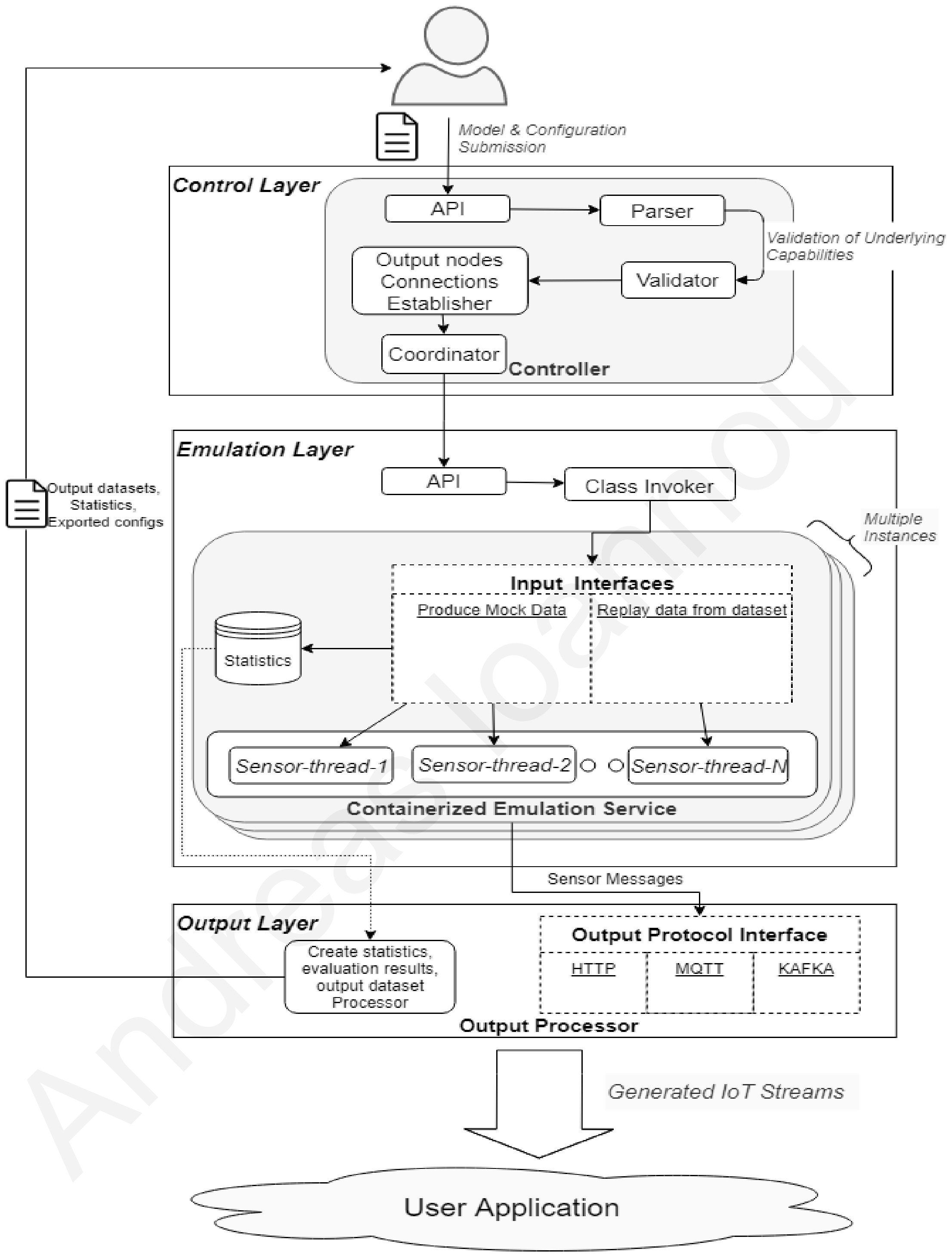


Figure 3.2.2 1: System High Level Architecture

Chapter 4

Model Mapping

4.1 Modelling of IoT Sensors

4.1.1 Input file, Sensor Prototypes & messages

4.1.2 Produce Mock data feature modelling

4.1.3 Replay data from dataset feature modelling

4.2 Output Mapping

In this chapter, the modelling of sensors, output and generally information for the input ***configs_file.json*** will be explained, along with file instances for real life scenarios applications.

In order to demonstrate more easily the input file attributes, let us consider a scenario in which we have a company, that wants to create and sell an application that aims to provide safety and security to the owner of a smart house. The application will be monitoring, and processing sensors' data produced by IoT devices in every room for a smart house and eventually will notify the owner of the house on time, if the house is under a physical or technical threat. Some threats for example would be a random fire at the house (could be detected through smoke detection and temperature sensor) or someone (maybe a thief) is in the house when the owner is not at the house (motion detection sensors). The idea here, is that all the needed sensors (motion detection, smoke detection, temperature sensors) will be constantly and continuously monitoring and populating the monitored data in order to represent the state of the house at any given time. The application on the other end, will be collecting and processing all the generated data stream from the sensors, and will have the logic to understand when the house is under a threat, or something is not ordinary. Due to the importance of the time needed for these continuous streaming of data to be processed (if there is a fire at the house, the user must be informed as soon as the processing detects that from the data), the company cannot afford to let the processing happen in the cloud (due to the latency that the data will need to be transferred), so it should place an intermediate node with enough resources to do the processing geographically closer to the house owner's region, using the fog computing concept and then the node would inform the user for any unfortunate event on time. This application could provide safety and security for the owner of the house at any time regardless if the owner is at home or not.

After the company has developed the application, of course it needs to test it through various scenarios, to evaluate the performance, and that indeed the user is being informed on time! For these testing scenarios, the company will need plenty of data to be produced according to each scenario or time period in the smart house. To this end, the company would be forced to go find and maybe rent, a house near to the fog node that the company owns, purchase all the above sensors, place them in the house, and program them or use them to provide data for each scenario. On top of that, every time that the company wants to test a different scenario, or use different types of sensors, it needs to do the same procedure again (remove the old sensors, purchase, locate and set up the new ones) for all the different testing scenarios. The long procedure of redeploying and adjusting each sensor for each testing scenario, results in waste of valuable effort, time and money.

It would be way easier, if the company could monitor and extract statistics for a single scenario, or at least create a basic estimation for the sensors, and provide this model to a workload generator, which will generate automatically data based on this input sensor model. With this approach, the company could just edit he input properties of the sensors models for each scenario, without the need to redeploy or do some extra effort to place or change the real sensor, which will result in less time consuming and cost-free solution. All the actions that the company should do is to find, create a sensor model, and provide this model description to the system. Of course, it will need some time to extract the sensors properties from real sensors, (in order to achieve this, maybe it should collect statistics or at least have an idea of the populating behaviour of the properties of the sensors). An example based on the fire scenario, the company could try to set real fire at some safe place, and monitor the average behaviour of the temperature sensors, by giving emphasis to extract the alternation of the temperature, in order to have an estimation of the range for this scenario. Then, it could provide these statistics to the workload generator model, which will also alter the population of the temperature sensors, accordingly to the statistics for the real fire. In addition, the company could also provide the exact real monitored temperature sensor from the fire experiment, and the system can replay them customized. This way, the company would only need to find and provide the input with the sensors descriptions, and the framework will output all the generated data straight to the company's application or node for faster process.

Having explained the tremendous benefits for the application users using such a tool, let us move deeper into how the users could provide the model description, according to their desired sensors properties.

4.1 Modelling of IoT Sensors

In this first chapter, all the sensor related mapping will be described, along with examples based on the above real-life scenario.

4.1.1 Input file, Sensor Prototypes & messages

In this section, we provide details about the modelling of the IoT devices and the sensors that the user should provide to the system. More precisely, the user needs to provide a JSON file called ***configs_file.json*** that includes the properties and the output messages' structures for every emulated IoT device and sensor. The structure of the JSON file starts with a root field, called *configs*, and one of the child fields of the *configs* root field, which is called *sensorDataConfigs* has all the information related to the sensor modelling. In this field a list of *sensorPrototypes* should be provided. A **SensorPrototype** is an abstract representation of an emulated sensor type / IoT device and holds information regarding sensors for the same type. For example, if we have 2 different sensors types (temperature sensor and pressure sensor), the system would require 2 sensorPrototypes, one of each sensor type. Currently, the system materializes two implementations for the SensorPrototype, namely, the **MockSensorPrototype** that represents artificially created sensors and generates IoT data based on user-provided patterns, and **DatasetSensorPrototype** that is able to replay an existing IoT dataset. Of course, the user can provide a mixture or a combination of the SensorPrototypes implementation in each scenario's input. We should note here that users are free to introduce new SensorPrototypes implementations by following the SensorPrototype abstraction class so as to implement user-driven functionalities in their emulation. In both prototypes, some attributes are common, like the *sensorPrototypeName*, which is mandatory for both of the materializations because is been utilized by the system as the identifier of the emulated device type, the *GenerationRate* that is the frequency of the data production and can be either static or dynamic, and the *sensorMessagePrototype* that provides information about the structure of the output messages. More precisely, **SensorMessagePrototype** stores the properties for each sensor message fields, message format, and populated values that the sensors will produce.

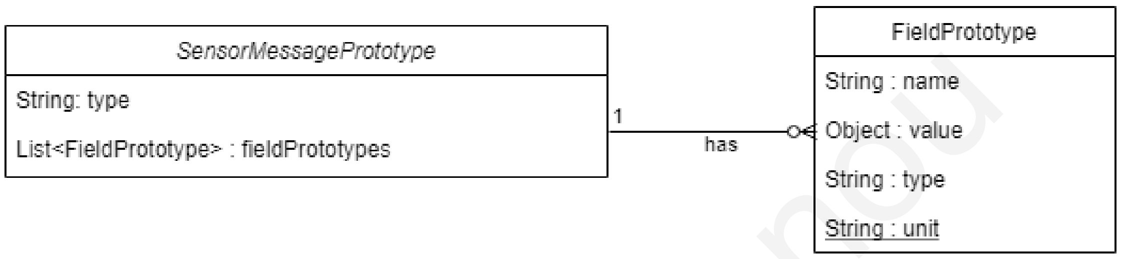


Figure 4.1.1 1: SensorMessagePrototype Structure

In Figure 4.1.1 1 we can see the structure of the **SensorMessagePrototype**. The *type* attribute states the type of the message format that the message will be constructed and sent to the output node. Currently, XML, JSON, TXT types are supported. Moreover, each sensor message can have multiple fields, hence the list *fieldPrototypes* is introduced. In a **FieldPrototype**, the user declares the *name* of the populated field, the populated value *type* (String, Boolean, Integer, Double, Object), information regarding the method of the future populated value, and optionally the *unit* for the populated values. What is more, a sensor message field value, can be created using some existing data population methods. The currently implemented data population methods are using *constant values*, *random number generation between number range*, following *normal distribution*, or finally following *probabilistic distributions*. The first 2 methods (constant values, random number between number range), were chosen because they correspond to the basic and most simple data population methods, that a user can use if he does not knows the exact sensor properties population behaviour yet, but he knows an approximation or a range of the values. On top of that, the system was enhanced with 2 more advanced and useful data population methods, the normal distribution (that uses mean and deviation) and probabilistic distributions (that uses key pair values of the actual value and the probability for this value to happen) in order to support data population that was exported using statistics. Of course, user can add more advanced data population methods, according to his needs.

Beyond the above-mentioned common properties each SensorPrototype subclass offers its own tailored properties which can be seen in the Figure 4.1.1 2

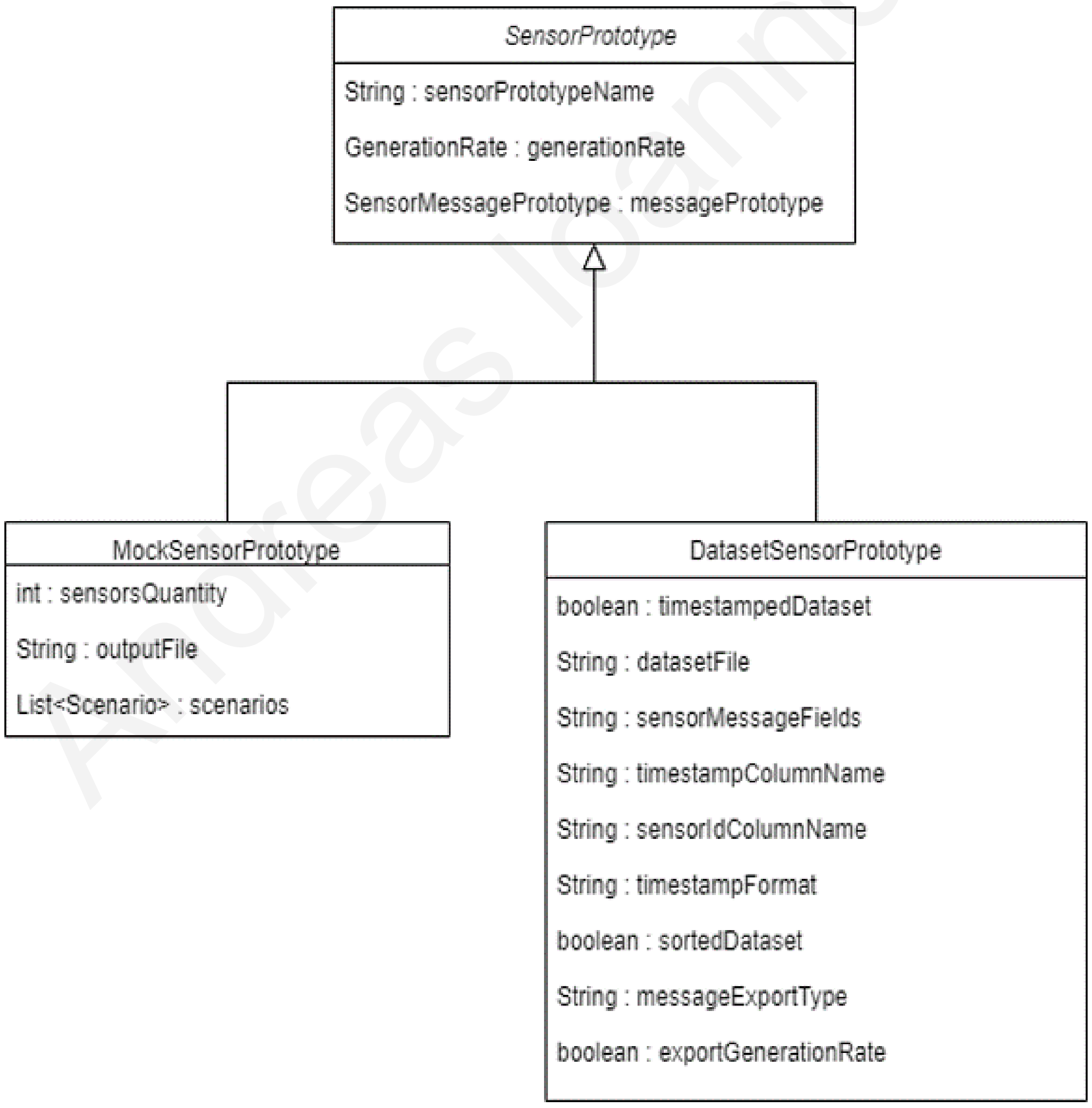
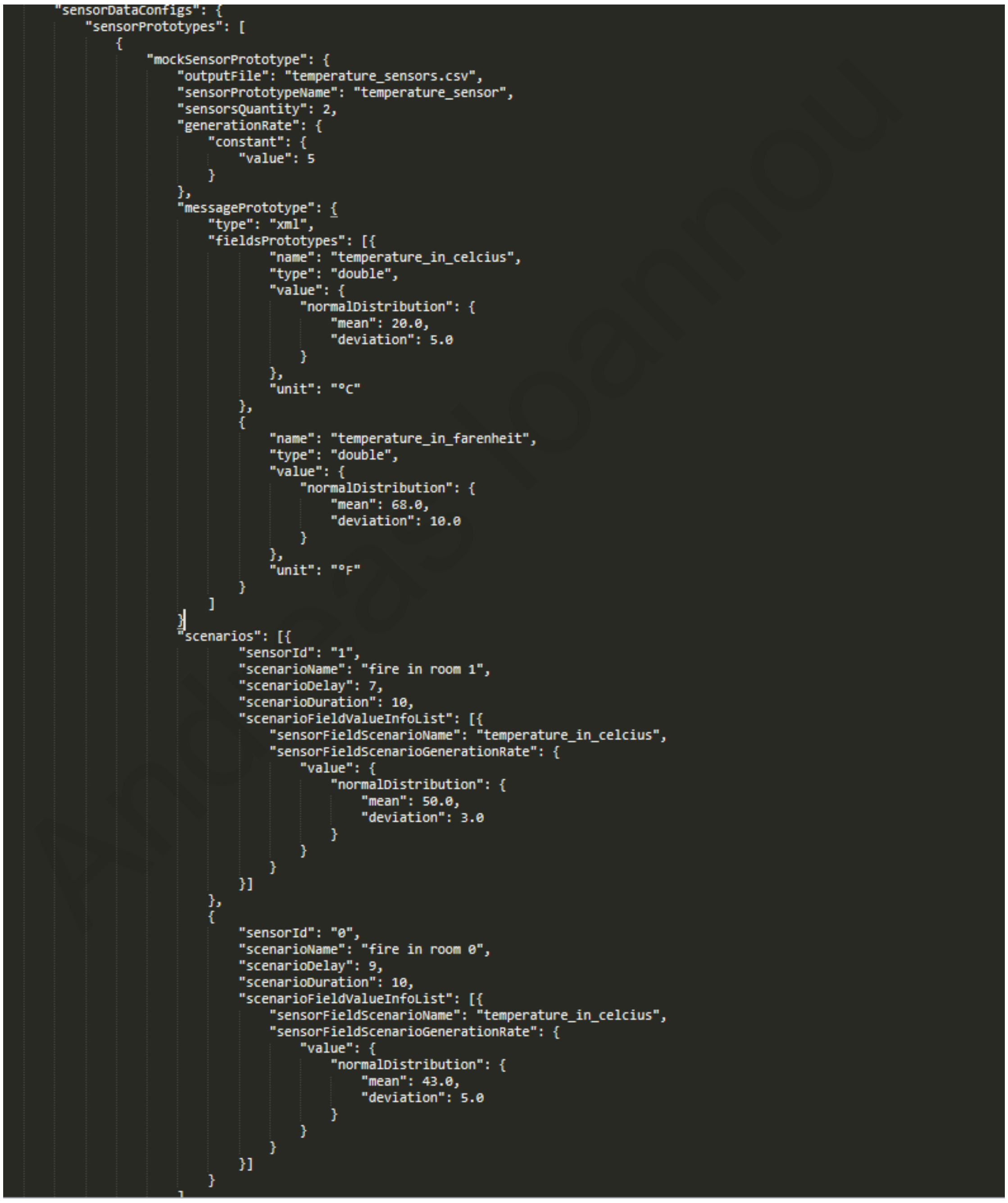

SensorPrototype
String : sensorPrototypeName
GenerationRate : generationRate
SensorMessagePrototype : messagePrototype
MockSensorPrototype
int : sensorsQuantity
String : outputFile
List<Scenario> : scenarios
DatasetSensorPrototype
boolean : timestampedDataset
String : datasetFile
String : sensorMessageFields
String : timestampColumnName
String : sensorIdColumnName
String : timestampFormat
boolean : sortedDataset
String : messageExportType
boolean : exportGenerationRate

Figure 4.1.1 2: SensorPrototype Model Diagram

4.1.2 Produce Mock data feature modelling

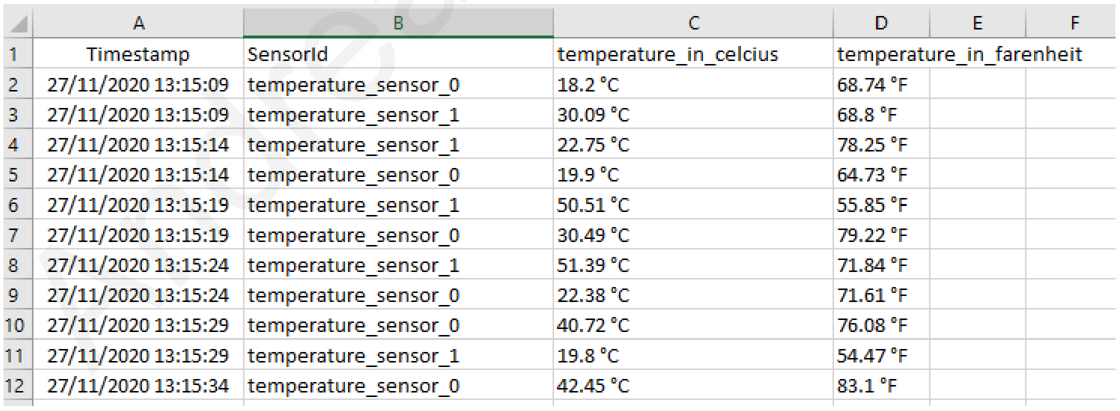
The implementation of the materialization of the SensorPrototype for the produce mock data by emulating sensors functionality, is the **MockSensorPrototype** . For this implementation, the user beside the common attributes that all sensorPrototypes share and stated above (*sensorPrototypeName*, *generationRate*, *messageSensorPrototype*) which are all mandatory for this materialization he should also provide some other runtime information about the emulated sensors that will be created. Initially, s*ensorsQuantity* defines the quantity of the emulated sensors that the system will create for the each specific MockSensorPrototype (for example if the MockSensorPrototype with sensorPrototypeName “temperature_Sensors” has sensorsQuantity = 2, it implies that the system will create 2 temperature sensors). Optionally, users are able to declare an output file name (*outputFile*) that all the produced data streams from sensors of the same type (same MockSensorPrototype) will be exported. In this way, users can perform post-analysis or repeat the experiment later. Finally, users are able to introduce *scenarios under each MockSensorPrototype*, which are simply time scheduled alterations of the running emulated sensors. With the *Scenarios* users perform more complex and realistic experiments without the need of redeployment of the whole emulation at the emulation phase. Each scenario has a *scenarioName* as identifier and will perform on the sensor with *sensorId of the MockSensorPrototype.* The *scenarioDelay* and the *scenarioDuration* identify the time that the scenario will be executed from the beginning of the experiment and the duration of it, respectively. The *scenarioFieldValueInfoList* depicts the effects and value changes that the sensors will experience at the duration of the scenario. Specifically, *scenarioFieldValueInfoList* is a list consisting of the fields (*sensorFieldScenarioName*) of the sensor that will change and their new value (*sensorFieldScenarioGenerationRate*) for the period of the scenario.



```
"sensorDataConfigs": {
    "sensorPrototypes": [
        {
            "mockSensorPrototype": {
                "outputFile": "temperature_sensors.csv",
                "sensorPrototypeName": "temperature_sensor",
                "sensorsQuantity": 2,
                "generationRate": {
                    "constant": {
                        "value": 5
                    }
                },
                "messagePrototype": {
                    "type": "xml",
                    "fieldsPrototypes": [{
                            "name": "temperature_in_celcius",
                            "type": "double",
                            "value": {
                                "normalDistribution": {
                                    "mean": 20.0,
                                    "deviation": 5.0
                                }
                            },
                            "unit": "°C"
                        },
                        {
                            "name": "temperature_in_farenheit",
                            "type": "double",
                            "value": {
                                "normalDistribution": {
                                    "mean": 68.0,
                                    "deviation": 10.0
                                }
                            },
                            "unit": "°F"
                        }
                    ]
                },
                "scenarios": [{
                        "sensorId": "1",
                        "scenarioName": "fire in room 1",
                        "scenarioDelay": 7,
                        "scenarioDuration": 10,
                        "scenarioFieldValueInfoList": [{
                            "sensorFieldScenarioName": "temperature_in_celcius",
                            "sensorFieldScenarioGenerationRate": {
                                "value": {
                                    "normalDistribution": {
                                        "mean": 50.0,
                                        "deviation": 3.0
                                    }
                                }
                            }
                        }]
                    },
                    {
                        "sensorId": "0",
                        "scenarioName": "fire in room 0",
                        "scenarioDelay": 9,
                        "scenarioDuration": 10,
                        "scenarioFieldValueInfoList": [{
                            "sensorFieldScenarioName": "temperature_in_celcius",
                            "sensorFieldScenarioGenerationRate": {
                                "value": {
                                    "normalDistribution": {
                                        "mean": 43.0,
                                        "deviation": 5.0
                                    }
                                }
                            }
                        }]
                    }
                ]
```

Figure 4.1.2 1: MockSensorPrototype Input Example

Figure 4.1.2 1 illustrates an example input for 1 MockSensorPrototype. In this input, we have the “temperature_sensor” MockSensorPrototype. The system will create 2 emulated temperature sensors, that each one will be generating new data per 5 seconds. All the output data stream of the 2 sensors will be exported to the “temperature_sensors.csv” file. Each sensor will be generating data, that consists of 2 fields (temperature_in_celcius, temperature_in_farenheit) and each generated message will be of XML format. These 2 fields will be having double values (temperature), following the normal distributions provided, along with their unit. Also, in this input we have 2 scenarios of fires that will take place in two different rooms. For instance, in the first scenario, the values of the temperature sensor 1 will be increased up to 50 Celsius degrees at the 7th second of the experiment for 10 seconds, while in the second scenario, the temperature of the sensor 0 will be increased up to 43 Celsius degrees starting from the 9th second of the experiment for 10 seconds. Figure 4.1.2 2 illustrates the initial generated data from temperature_output.csv file, based on the input in Figure 4.1.2 1 .



	A	B	C	D	E	F
1	Timestamp	SensorId	temperature_in_celcius	temperature_in_farenheit		
2	27/11/2020 13:15:09	temperature_sensor_0	18.2 °C	68.74 °F		
3	27/11/2020 13:15:09	temperature_sensor_1	30.09 °C	68.8 °F		
4	27/11/2020 13:15:14	temperature_sensor_1	22.75 °C	78.25 °F		
5	27/11/2020 13:15:14	temperature_sensor_0	19.9 °C	64.73 °F		
6	27/11/2020 13:15:19	temperature_sensor_1	50.51 °C	55.85 °F		
7	27/11/2020 13:15:19	temperature_sensor_0	30.49 °C	79.22 °F		
8	27/11/2020 13:15:24	temperature_sensor_1	51.39 °C	71.84 °F		
9	27/11/2020 13:15:24	temperature_sensor_0	22.38 °C	71.61 °F		
10	27/11/2020 13:15:29	temperature_sensor_0	40.72 °C	76.08 °F		
11	27/11/2020 13:15:29	temperature_sensor_1	19.8 °C	54.47 °F		
12	27/11/2020 13:15:34	temperature_sensor_0	42.45 °C	83.1 °F		

Figure 4.1.2 2: Sample output for the input in *Figure 4.1.2 1*

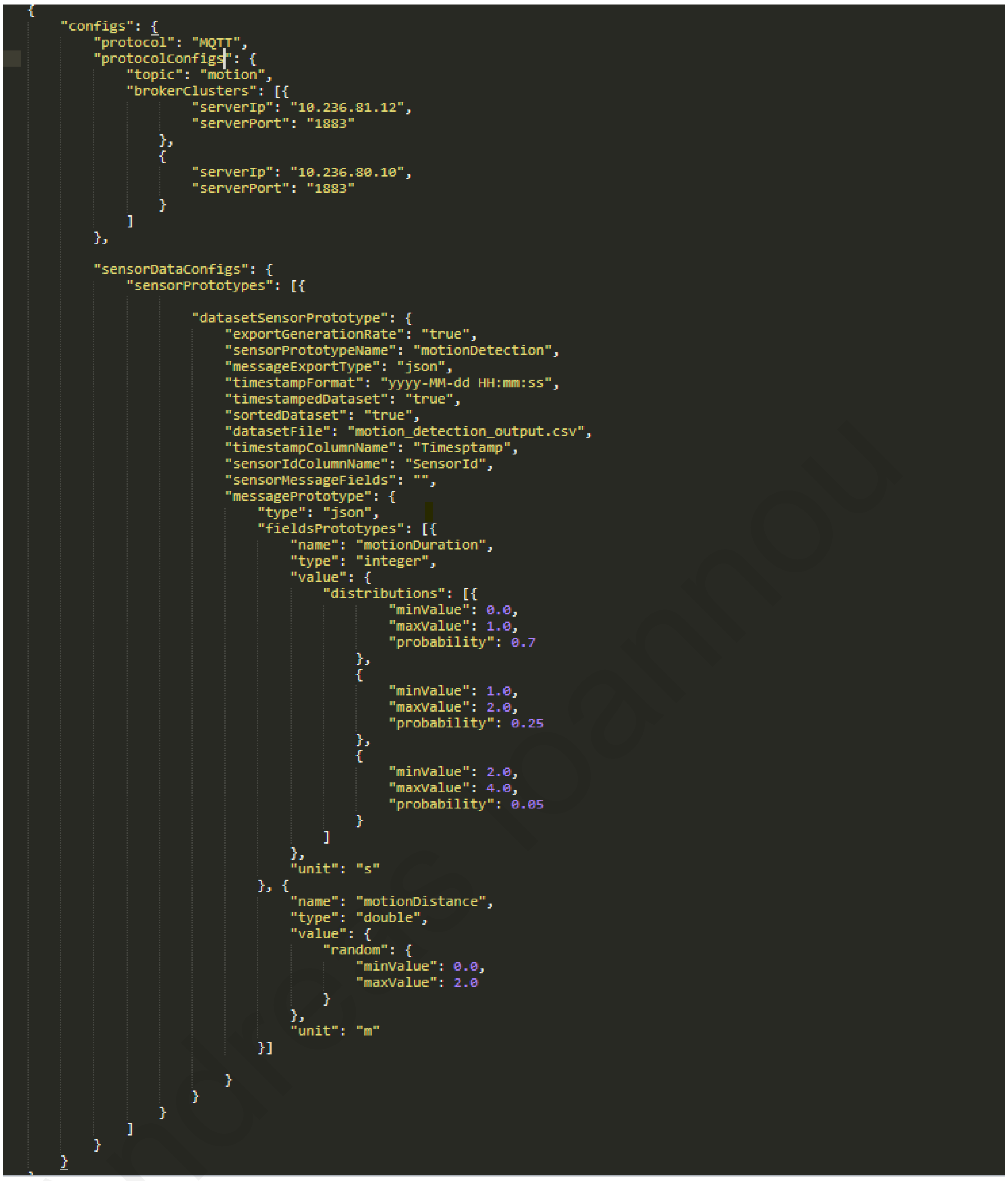
4.1.3 Replay data from dataset feature modelling

For the other main functionality of the system, the ability to replay sensor data form dataset, the implementation materialization of SensorPrototype is called **DatasetSensorPrototype.** Here, users need to provide a DatasetSensorPrototype (under the *sensorPrototypes* field inside the ***configs_file.json***) for each dataset (real sensor exported data). Each CSV *datasetFile*, will have recorded real sensor messages for each sensor prototype individually, that they will be replayed from the system including the ability to be expanded. The dataset should have real sensor messages in each row - record (every column will represent the value for each field) that will be attempted to be replayed using the timestamp column (*timestampColumnName*) of the row (sensor message) if the dataset is being timestamped (*timestampedDataset*), or user's preferences (*generationRate*) can be used, for each sensor message (records of dataset) time interval. In the first line of the CSV dataset, the names for each field (column) will be declared in order for the system to be able to do some more advanced and customizable processing for the data. Some more features for this input method are that the system can also sort the dataset using external sorting [4] if the dataset is not sorted (declared by *sortedDataset*) using a timestamp field and format (*timestampFormat*) provided in the configuration file and can send specific fields from each record (*sensorMessagesFields*) that the user desires (maybe all of them, none or some of them separated by comma). In addition, users can add more fields, so that the system can generate mock data using the messagePrototype field inherited from the SensorPrototype. These extra fields will be created on runtime. For datasets that

[4] https://mvnrepository.com/artifact/com.google.code.externalsortinginjava/externalsortinginjava/0.4.5

support exported messages for multiple sensors (of the same type) each sensor message can be mapped to the specific sensorId using the *sensorIdColumn*. Finally, if *exportGenerationRate* is set as true, the system will export the characteristics of the DatasetSensorPrototype, as a ready MockSensorPrototype configuration, so that the user can provide it to the other input feature to create mock data for the sensor type. With all these capabilities, users can replay data from CSV dataset, with customized features (add more fields, send less fields, edit the generation rate etc.) and test her application with real dataset with the ability to edit or adjust it for multiple desired scenarios.

In Figure 4.1.3 1 the we can see an example of input for DatasetSensorPrototype. The instance is regarding the "*motionDetection"* sensors. Before moving to the next fields, let us have a look at some rows of the input dataset file which will be replayed ("motion_detection_output.csv") which can be seen in Figure 4.1.3 2. In the rows of the CSV file, we can see the recorded messages that the company for example already exported from real sensors. The recorded messages have one field called "*movement"* with boolean values.

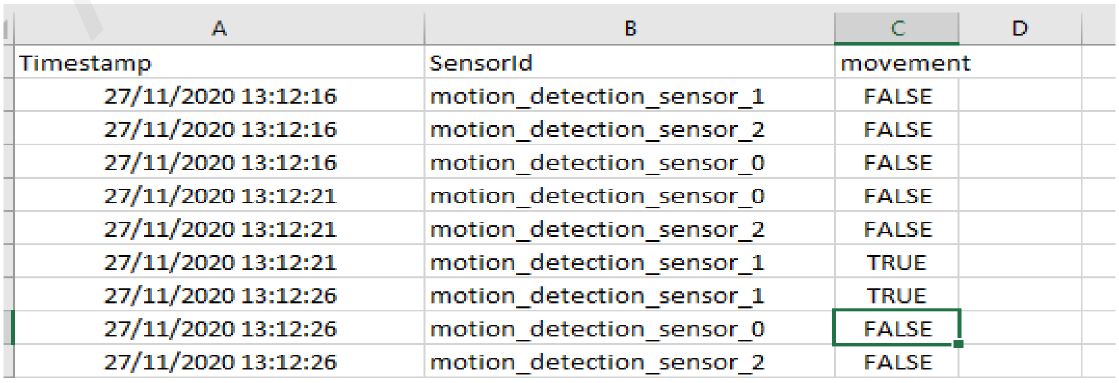


```
{
    "configs": {
        "protocol": "MQTT",
        "protocolConfigs": {
            "topic": "motion",
            "brokerClusters": [{
                    "serverIp": "10.236.81.12",
                    "serverPort": "1883"
                },
                {
                    "serverIp": "10.236.80.10",
                    "serverPort": "1883"
                }
            ]
        },

        "sensorDataConfigs": {
            "sensorPrototypes": [{

                    "datasetSensorPrototype": {
                        "exportGenerationRate": "true",
                        "sensorPrototypeName": "motionDetection",
                        "messageExportType": "json",
                        "timestampFormat": "yyyy-MM-dd HH:mm:ss",
                        "timestampedDataset": "true",
                        "sortedDataset": "true",
                        "datasetFile": "motion_detection_output.csv",
                        "timestampColumnName": "Timesptamp",
                        "sensorIdColumnName": "SensorId",
                        "sensorMessageFields": "",
                        "messagePrototype": {
                            "type": "json",
                            "fieldsPrototypes": [{
                                "name": "motionDuration",
                                "type": "integer",
                                "value": {
                                    "distributions": [{
                                            "minValue": 0.0,
                                            "maxValue": 1.0,
                                            "probability": 0.7
                                        },
                                        {
                                            "minValue": 1.0,
                                            "maxValue": 2.0,
                                            "probability": 0.25
                                        },
                                        {
                                            "minValue": 2.0,
                                            "maxValue": 4.0,
                                            "probability": 0.05
                                        }
                                    ]
                                },
                                "unit": "s"
                            }, {
                                "name": "motionDistance",
                                "type": "double",
                                "value": {
                                    "random": {
                                        "minValue": 0.0,
                                        "maxValue": 2.0
                                    }
                                },
                                "unit": "m"
                            }]
                        }
                    }
                }
            ]
        }
    }
```

Figure 4.1.3 1: DatasetSensorPrototype Input Example



Timestamp	SensorId	movement
27/11/2020 13:12:16	motion_detection_sensor_1	FALSE
27/11/2020 13:12:16	motion_detection_sensor_2	FALSE
27/11/2020 13:12:16	motion_detection_sensor_0	FALSE
27/11/2020 13:12:21	motion_detection_sensor_0	FALSE
27/11/2020 13:12:21	motion_detection_sensor_2	FALSE
27/11/2020 13:12:21	motion_detection_sensor_1	TRUE
27/11/2020 13:12:26	motion_detection_sensor_1	TRUE
27/11/2020 13:12:26	motion_detection_sensor_0	FALSE
27/11/2020 13:12:26	motion_detection_sensor_2	FALSE

Figure 4.1.3 2: Motion_detection_output.csv instance

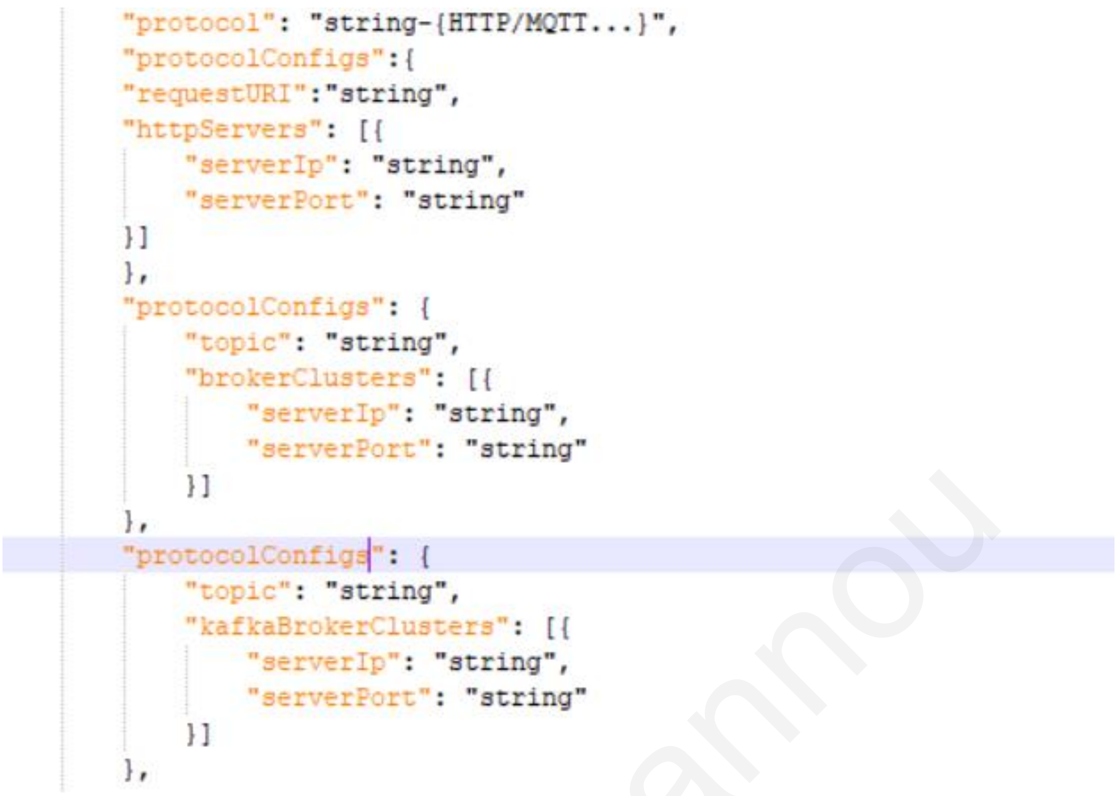
In the input instance (Figure 4.1.3 1), we can see firstly that the dataset is timestamped (*timestampedDataset* is true) and the dataset is already sorted (*sortedDataset* is true). This implies that, when the dataset will be attempted to be replayed, each sensor message (each row) will be reproduced based on the timestamp column with name “Timestamp” (*timestampColumnName*) of each row. The first message to be replayed (the first row with data in this case), will acquire the current date, and each following sensor message will be replayed based on the difference of the first row, and the difference of the timestamps of the 2 continuous rows of the dataset. So for example, if the first row was replayed at time 18:22:12 from the workload generator, this means that the next message will be replayed at the same time (13:12:16 -13:12:16 = 0 seconds) but the 4th sensor message with timestamp 13:12:21 will be replayed after 5 seconds (|| 13:12:16 -13:12:21 || = 5 seconds), which means at 18:22:17 time of the workload generator. On top of that, it is visible that each message reproduction will be for the specified sensor in the column “*sensorID*” and that none of the fields of the CSV file will be used in the new replayed messages (*sensorMessageFields* is empty). Instead, each replayed message will be enhanced with mock fields that will be generated at runtime using the messagePrototype (Field “motionDuration” will follow probabilistic distribution, with 70 % probability to generate an integer between [0 .. 1], 25% probability to generate integer between [1 .. 2] and 5 % probability to generate an integer between [2 .. 4] - Field “motionDistance” will be populating double random values between the range [0.0 , 2.0]) and each reproduced message will be of JSON format.

4.2 Output Mapping

In the input *configs_file.json* file, users should also provide information regarding the output protocols and destinations that the generated streams will be sent to. Besides the Sensor input implementations, and the different messages type format supported (JSON, XML, TXT) the system supports an extensible interface for output protocols. Currently supports 3 output message protocols (HTTP, MQTT and using KAFKA broker) and their configurations' implementation, but users can easily expand the system to support even more protocols by just implementing the output sensor protocol interface.

Regarding the output protocol information, the user must first declare the output *protocol* under the *configs* root field, inside the input **configs_file.json** that the messages will be sent with to the output nodes. Beyond that, users must provide the exclusive configurations for the declared protocol under *protocolConfigs* along with output nodes information.

In the Figure 4.2 1 we can see all the output information that the user must provide. The *protocol* is mandatory and must be one from the supported implementations and below we can see the configs needed for each protocol. (user should provide one key with *protocolConfigs* matching the output protocol's required information)



```
"protocol": "string-{HTTP/MQTT...}",
"protocolConfigs":{
"requestURI":"string",
"httpServers": [{
    "serverIp": "string",
    "serverPort": "string"
}]
},
"protocolConfigs": {
    "topic": "string",
    "brokerClusters": [{
        "serverIp": "string",
        "serverPort": "string"
    }]
},
"protocolConfigs": {
    "topic": "string",
    "kafkaBrokerClusters": [{
        "serverIp": "string",
        "serverPort": "string"
    }]
},
```

Figure 4.2 1 :Output Protocol Configs

User must provide the corresponding configs for the protocol he declared under protocolConfigs. For the HTTP protocol, the user must provide the request endpoint (*requestURI*) that the messages will be sent to. Besides that, user should provide a list for *httpServers* that each sensor message request will be attempted to be sent to. For each httpServer the *serverIp* and *serverPort* are required. For the supported publish-subscribe protocols, the configs required are the same. For MQTT and KAFKA protocol, users must provide a list of *brokerClusters* details (list of MQTT broker *serverPort* and *serverIp*), and the for the Kafka related configs a list of *kafkaBrokerClusters* respectively. Finally, for both publish-subscribe protocols, the *topic* should be provided.

Chapter 5

Implementation Details

5.1 Main Users actions

5.1.1 Start of Workload Generator

5.1.2 Stop of Workload Generator

5.1.3 Restart of Workload Generator

5.2 Extendable Interfaces Overview

5.2.1 Output Protocol Interface

5.2.2 SensorPrototype Input Interface

5.3 Produce Mock Sensor Data Implementation

5.3.1 Components and Association

5.3.2 Class Modeling

5.3.3 Flow

5.4 Replay sensor data from Dataset Implementation

5.4.1 Components and Association

5.4.2 Dataset Thread Modeling

5.4.3 Flow

5.5 Output protocol Implementations

5.6 API Endpoints

5.6.1 Workload Generator Actions APIs

5.6.2 Sensor Actions APIs

In this chapter more technical information will be provided, like how the system internally works, how the extendable interfaces are connected to the system and what a developer needs to do to extend them, what are the exact procedures for each input and output currently implemented interface, as well as an introduction to the API endpoints that the system currently supports and their purpose.

5.1 Main Users actions

In this first section of this chapter, the main actions that the user can do or request from the workload generator will be introduced, along with the workload generator states and each action's workflow.

The basic actions that the user will be doing to the workload generator as soon as it is up and ready for execution, are to *start* (could mean initiate/initialize or resume), *stop* or *restart* it.

These actions have as a result for the workload generator to be in a different state at any time. In the Figure 5.1 1 below we can see how each of these actions affect the states of the workload generator. The continuous arrows imply that one action is needed for the workload generator to move to the next state, and the dashed lines, meaning that no action is required to move to this state. Let us briefly explain the below state diagram, before moving to each action workflow. Once the workload generator container is deployed, the workload generator state will be ***Ready***. From this state (after the deployment) the only action that the user can trigger is the *start*, where in this state it can be considered as *initiate/initialize*, since it will cause the workload generator to be initiated and initialized by passing from the ***Initialized*** state, and then without further action start the workload generator, changing its state to ***Started***. From the ***Started*** state, the only action allowed is the *stop* action, which will make the workload generator to be paused temporarily causing it to change state to

Paused. There are 2 possible state transitions for the workload generator once being in ***Paused*** state. If the user apply the *start* action, which in this case can be considered as *resume*, will force the workload generator to resume from where it left, or the user can trigger the *restart* action, that will terminate all the pending or paused procedures of the workload generator, and completely causing it to restart from the beginning, by passing from the ***Stopped***, ***Initialized*** and subsequently ending once again to the ***Started*** state again.

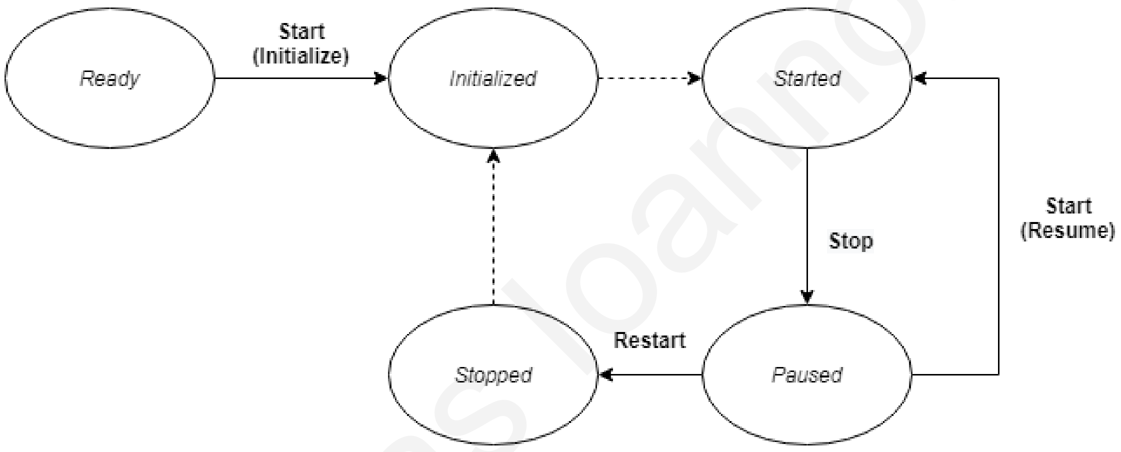


Figure 5.1 1: State Transition diagram for Workload Generator

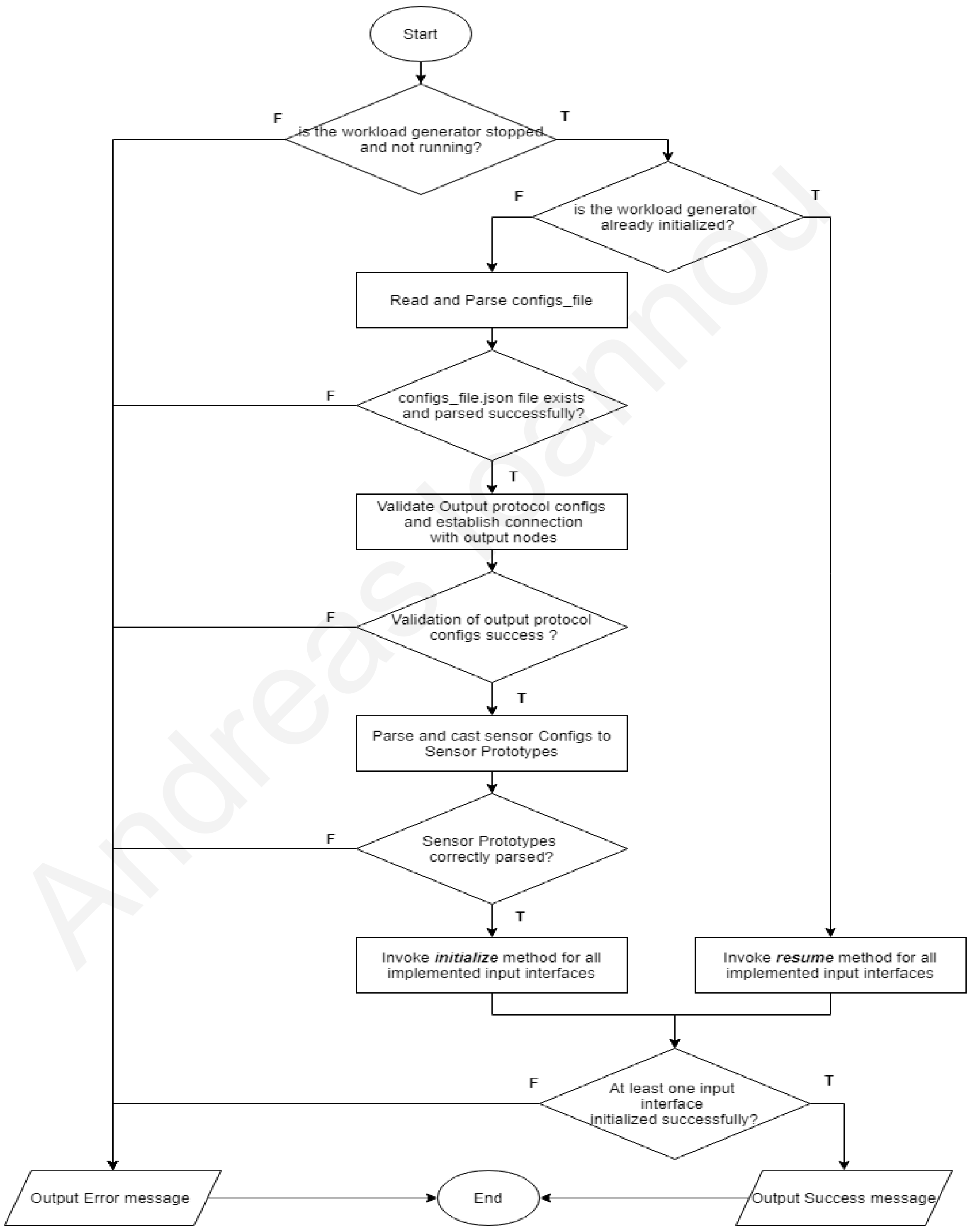
Now let us move on to explain in depth the flow and the internal procedures of each user action.

5.1.1 Start of Workload Generator

When the workload generator container is up and ready to be used, the user can only do one action, to start it by executing the **POST *****/workloadGenerator/start*** API. In the Figure 5.1.1 1 we can see the flow diagram for the *start* user action. Once this API is executed, which implies that the configs and the input files are already located in the declared directory, the first thing that the workload generator will do is to verify that it is not already running. If it is, it will output an error to a user that the workload generator is already started. If it is not running, it will check whether it is already initialized and paused, or has not been initiated yet.

If it is not initiated yet, it will try to read and parse the *configs_file.json*. If the input file exists in the declared path, is a valid json, and has the *configs* as root field, it will move on to validate the output protocol and its *protocolConfigs* (as explained in the section 4.2) by invoking the "*validateAndProcessConfigs*" method for the output protocol interface implementation matching the provided protocol, and if the mandatory fields for each protocol are provided, and the protocol is supported, it will invoke the "*initializeConnections*" method for the output protocol interface implementation based on the protocol, to establish connections with the output nodes. Then, the system will try to read the sensorDataConfigs, and cast the sensorPrototypes list, each sensorPrototype to the according type (datasetSensorPrototype, mockSensorPrototype), leaving the internal validations for each different implementation on the implementation level. Final step, before the flow ends, the system will invoke the "*initiate*" implemented method for each sensorPrototype implementation that is provided in the sensorDataConfigs (each input method discussed in section 4.1), and if at least one of the implementations validate accordingly their input sensorPrototypes without any error and start the emission of sensor messages, the workload generator will be considered *initialized*, and will output success to the user through the API. If none of the implementations has been successfully initialized, it will be considered not initialized and will output the error to the user.

On the other hand, if the workload generator is initialized but not running (which is the case that it was running and then it was stopped), the system will invoke the "*resume*" implemented interface method for each initialized sensorPrototype input, and again verify that at least one of the implementations resumed the sensor data generation successfully, otherwise will output that the workload generator could not be started to the user. For each failure case in the flow the system will try to output according error to the user.



Start
is the workload generator stopped and not running?
F
T
is the workload generator already initialized?
F
T
Read and Parse configs_file
configs_file.json file exists and parsed successfully?
F
T
Validate Output protocol configs and establish connection with output nodes
Validation of output protocol configs success ?
F
T
Parse and cast sensor Configs to Sensor Prototypes
Sensor Prototypes correctly parsed?
F
T
Invoke initialize method for all implemented input interfaces
Invoke resume method for all implemented input interfaces
At least one input interface initialized successfully?
F
T
Output Error message
End
Output Success message

*Figure 5.1.1 1: Flow Diagram for **Start** User Action*

5.1.2 Stop of Workload Generator

The user can at any time pause or stop the workload generator (if it is running) with the API. For this flow, the system will verify that the workload generator is already started, if it is, then it will directly invoke each running input implemented interface “*pause*” method, where the goal for all the input interfaces, is to pause the data generation and all pending tasks that each one of them is running, until maybe the user will call the start again so the data generation will be resumed.

5.1.3 Restart of Workload Generator

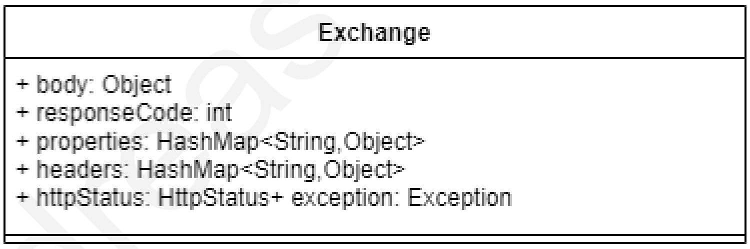
This action is not the same as the resume action (the workload generator starts again after the stop), and also can be triggered only when the workload generator is *Paused* (which implies it is initialized), otherwise it will output an error message. The purpose of this action is for the user to have the ability to change the input *configs_file.json* (maybe alter the output protocol or the sensorPrototypes for another test scenario and generally different input) and start again the workload generator from the beginning with the new input.

The restart procedure starts with the user calling the **POST** ***/workloadGenerator/restart*** API which means that the system firstly sets the workload generator state as not initialized, and then invoking the “*terminate*” method for all of the running input interface implementations. In the continuation, it will also trigger the termination for the connections with the output protocol nodes, by calling the “*terminate*” method of the provided output protocol interface implementation. Finally, it will do exactly the same algorithm as the Start action (read and parse again the configs, parse the *sensorDataConfigs* and create the new sensorPrototypes, establish connections with maybe the new output nodes or output protocol, and start the generation with the new input configs_file.json)

5.2 Extendable Interfaces Overview

After we introduced some methods for the implemented interfaces, it is time to check the interfaces in more depth, in order for any developer to be able to extend them with their own implementation.

Before we go deeper with the interfaces and the implementations let me introduce a model I created, that is being used and passed on the implementations methods that the developer needs to use and to be aware of, which is called "*Exchange*".



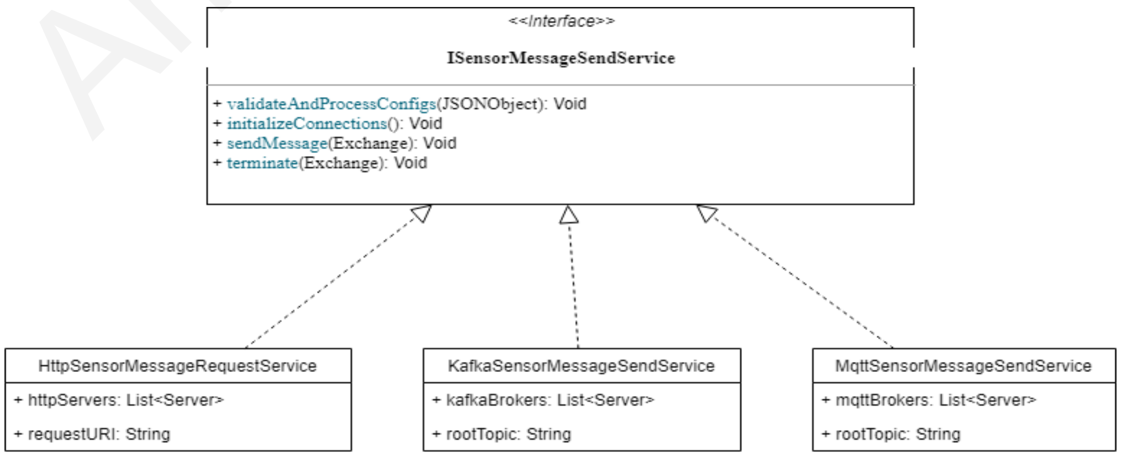
```
Exchange
+ body: Object
+ responseCode: int
+ properties: HashMap<String,Object>
+ headers: HashMap<String,Object>
+ httpStatus: HttpStatus+ exception: Exception
```

Figure 5.2 1: Exchange Date Model

Figure 5.2 1 above, is the representation of the fields of the class **Exchange**. It is a flexible model that helps the developer pass information without constraints, and uses it also to manipulate the API response, that the user would like to see when triggering each user action. This object is being created and initialized once a new API call starts and is deleted once the API call finishes. It is easier to consider this model, as the exchange of information between methods, that are being passed and enhanced accordingly from method to method. Firstly, the *body*, *responseCode*, *httpStatus* fields, are the fields that the developer does not have to edit (if he will, they will be overridden either way in the Controller when the API call finishes) and represent the response body and the HTTP Response Code, that will be returned to the user through the API response. So basically, the developer can use the properties Map, or the headers map (that are key value pairs), to pass information from method to method if he wants to. He can set a property for example like *exchange.setProperty("FLAG", true),* where the first parameter is the property name (*String*) and the second it is the actual property value (in this example *boolean* with value *true*), and he can retrieve the property value for this property in a later method by calling *exchange.getProperty("FLAG", Boolean.class),* that will return the value as *boolean* (or null if the value is not being set as *boolean* when the setProperty method was called), or exchange.getProperty("FLAG") that will return the object value of the property (in this case the developer has to cast it)

5.2.1 Output Protocol Interface

In this section, we will introduce the output protocol interface, its current implementations and how it can be implemented to support more custom extensions.



<<Interface>>
ISensorMessageSendService
+ validateAndProcessConfigs(JSONObject): Void
+ initializeConnections(): Void
+ sendMessage(Exchange): Void
+ terminate(Exchange): Void
HttpSensorMessageRequestService
+ httpServers: List<Server>
+ requestURI: String
KafkaSensorMessageSendService
+ kafkaBrokers: List<Server>
+ rootTopic: String
MqttSensorMessageSendService
+ mqttBrokers: List<Server>
+ rootTopic: String

Figure 5.2.1 1: ISensorMessageSendService Interface along with its current implementations class model diagram

In the Figure 5.2.1 1 above we can see the output interface, with name “***ISensorMessageSendService***”. This interface represents the way that the messages will be sent to the output node, and more specifically through which message transfer protocol or channel. Currently, there are 3 implementations, one for each of the output supported protocols (HTTP, MQTT, using KAFKA). All of the current implementations have implemented the 4 methods that the interface provides, and can of course have some extra exclusive fields that represent the protocol more accurately (e.g.: rootTopic for pub/sub protocols).

The first method, which will be the one to be called first from the *Controller*, is the *validateAndProcessConfigs*(JSONObject protocolConfigs). This method’s purpose is for each implementation to validate accordingly that the *protocolConfigs* passed as a JSONObject from the **Controller** (the input *protocolConfigs* the user provided in the *configs_file.json*), are the required and needed ones, based on each protocol (e.g.: *requestURI* for HTTP protocol and for the pub/sub protocols the *rootTopic* that the messages will have when being sent to the brokers).

Then, the *initializeConnections*() method will be invoked in order to initialize any needed configuration and establish connections with the user desired output nodes, and do the required processing based on the protocol, in order to be ready to send messages once they arrive (e.g.: pub/sub protocols to create Producer Clients). Once this method is finished successfully, it would imply that the implementation is ready to handle messages in order to send them to the already established connections with the output nodes.

The next and most important method is the *sendMessage* (String sensorId, String message, SensorMessageEnum contentType). This method's purpose, as the name implies, is to create the end packaged message and send it to the required destination, using the exclusive output protocol for the implementation. The arguments for this method are the populated sensor message, along with the sensorId and finally the message ContentType, in order to construct the message accordingly and send the packaged message to the desired server.

Finally, the last method that each output protocol implementation has to implement, is the *terminate*(). This method will be called from the **Controller** for each implementation, only when the user action is the *restart*, in order to terminate any pending task (e.g.: terminate ProducerClients for pub/sub protocols), and close smoothly the connections with the output machines (servers, or broker servers).

Any developer that wants to extend the interface, needs to give a new implementation by implementing the above-mentioned methods, and add his new Protocol in **OutputProtocolEnum** enumeration class.

5.2.2 SensorPrototype Input Interface

As stated in Chapter 4, each sensorPrototype can be used along with an input method (to emulate sensors of a mockSensorPrototype and produce mock data, or to replay the data for the sensors of a datasetSensorPrototype from a CSV dataset). Both these 2 input methods are implementations of an interface, which can be extended in order for the user to create even more new input methods easily. It is important to mention that all the sensorPrototypes that share the same input method, will be used only by the implementation of the input method they are declared to.

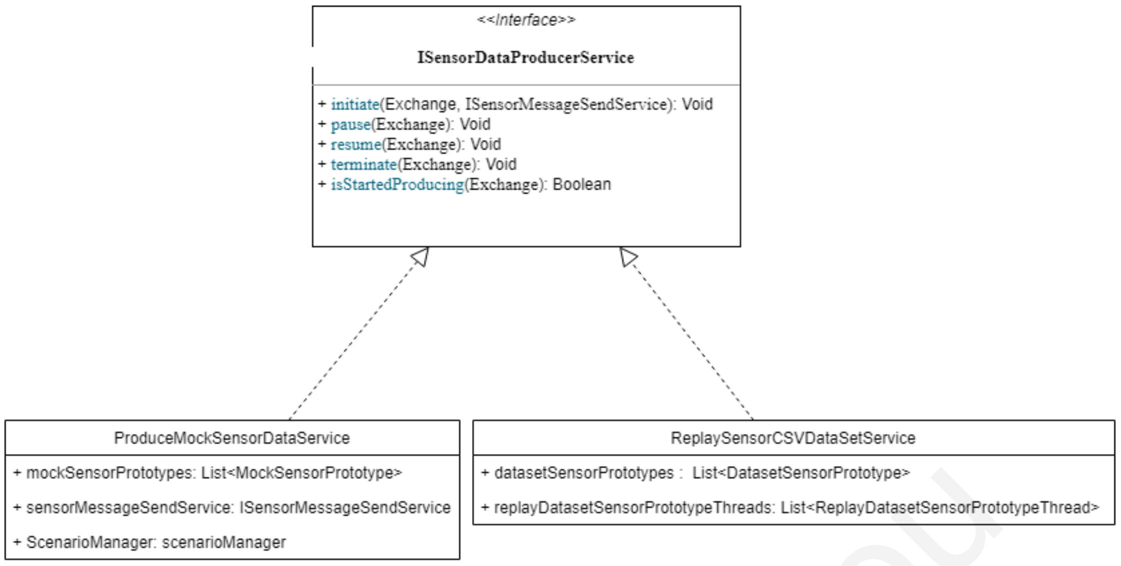

<<Interface>>
ISensorDataProducerService
+ initiate(Exchange, ISensorMessageSendService): Void
+ pause(Exchange): Void
+ resume(Exchange): Void
+ terminate(Exchange): Void
+ isStartedProducing(Exchange): Boolean
ProduceMockSensorDataService
+ mockSensorPrototypes: List<MockSensorPrototype>
+ sensorMessageSendService: ISensorMessageSendService
+ ScenarioManager: scenarioManager
ReplaySensorCSVDataSetService
+ datasetSensorPrototypes : List<DatasetSensorPrototype>
+ replayDatasetSensorPrototypeThreads: List<ReplayDatasetSensorPrototypeThread>

Figure 5.2.2 1: ISensorDataProducerService Interface with its current implementations diagram

In the Figure 5.2.2 1 we can see the class diagram for the input method interface, along with the current implementations. The input method interface called "***ISensorDataProducerService***" has currently 5 methods that each implementation must override.

Firstly, the *initiate*(Exchange exchange, ISensorMessageSendService sensorMessageSendService) method is the first one to be called from the **Controller** (in the start user action) that does the most important work, start generating sensor data stream according to its implementation logic. It should be noticed that the second parameter (sensorMessageSendService) for this method passed from the Controller, along with the exchange, is the implementation for the output protocol, that is already initialized and ready to be used. Hence the developer needs to use this implementation when he will be sending the messages, by calling the *sendMessage* (String sensorId, String message, SensorMessageEnum contentType) method for every sensor message that his implementation will generate. Thus, in this method, all the mandatory processing needs to happen, in order when the method finishes, it would be meaning that all the sensorPrototypes that use this input method, have already been validated, and processed, and their sensors are triggered to generate a data stream using the provided protocol implementation.

A way to tell if the emission of the sensor messages started successfully to the **Controller**, in order to inform the user, is the method *isStartedProducing*(). This method, as the name suggests, it returns true if the emission of data stream started successfully, or false if something went wrong, in order for the Controller, to count how many of the implementations started, and inform the user if at least one started successfully that the workload generator is running (errors in the logs will be seen for the implementation that failed to start producing data)

Moving on, when the user trigger the *stop* action, the *pause*(Exchange exchange) method will be called for each implementation, in order to temporarily stop all the data production (might also pause the created threads), with a way that in the future, if the user attempts to resume the data production, it will be feasible to start from where it is left.

The way that the user can trigger the *resume* of the production of data, is by triggering the *start* action, as it is mentioned above, but this time, the method that will be called it would be the *resume*(Exchange exchange) instead of the initiate (because the workload generator is already initialized, and just paused). Like the name implies, the implementation will use this method in order to start the production of data, from where it left when the *pause* method was called.

Finally, when the user triggers the *restart* action, it is time to close all the pending tasks and clear all the sensors for the implementation, by using the method *terminate*(Exchange exchange). After the execution of this method, the system will consider the implementation as done, because it might never be used again (if the user does not provide new sensorPrototypes with the implementation input method), hence all the sensors or threads, should be destroyed and free the space accordingly for future use.

5.3 Produce Mock Sensor Data Implementation

In this section, the flow of the Produce Mock Data implementation for the input method interface will be introduced and explained, along with some technical details for its implementation.

5.3.1 Components and Association

Before we go deeper with the implementation, let us transform the high-level workload generator

Figure 3.2.2 *1* flow from section 3.2.2 by replacing the emulation layer, with the current emulation instance implementation.

In the Figure 5.3.1 1 we can see the workload generator flow, that represents the scenario for this emulation instance. The user will provide the configs_file.json with the mockSensorPrototypes, and the Controller will do all the processing as the previous figure, and pass to this emulation instance implementation, the MockSensorPrototypes, along with the desired and initialized output protocol interface implementation. By observing the emulation layer for this instance, we see the created sensor threads, along with 2 new components, the WriterThread and the Scenario Manager. The WriterThread is the thread that has the job to receive sensor messages from the sensor threads to export them to the output files. The

ScenarioManager component coordinates and monitors the management of the scenarios (scheduling, and execution using one thread per scenario - more on that later). Now that we introduced the new components, it is easier to have a clearer view of the underlying architecture for this emulation instance. The emulation coordinator creates and initializes the WriterThread and the ScenarioManager and communicates with them based on the user actions. The most important functionality of course for the emulation coordinator, is to create and start the sensor threads, in order for the generation of the stream to be started. The output layer acts like it was explained in the previous initial Figure 3.2.2 *1*, sending the generated stream to the user application using the output protocol interface implementation and all the output files and statistics to the user.

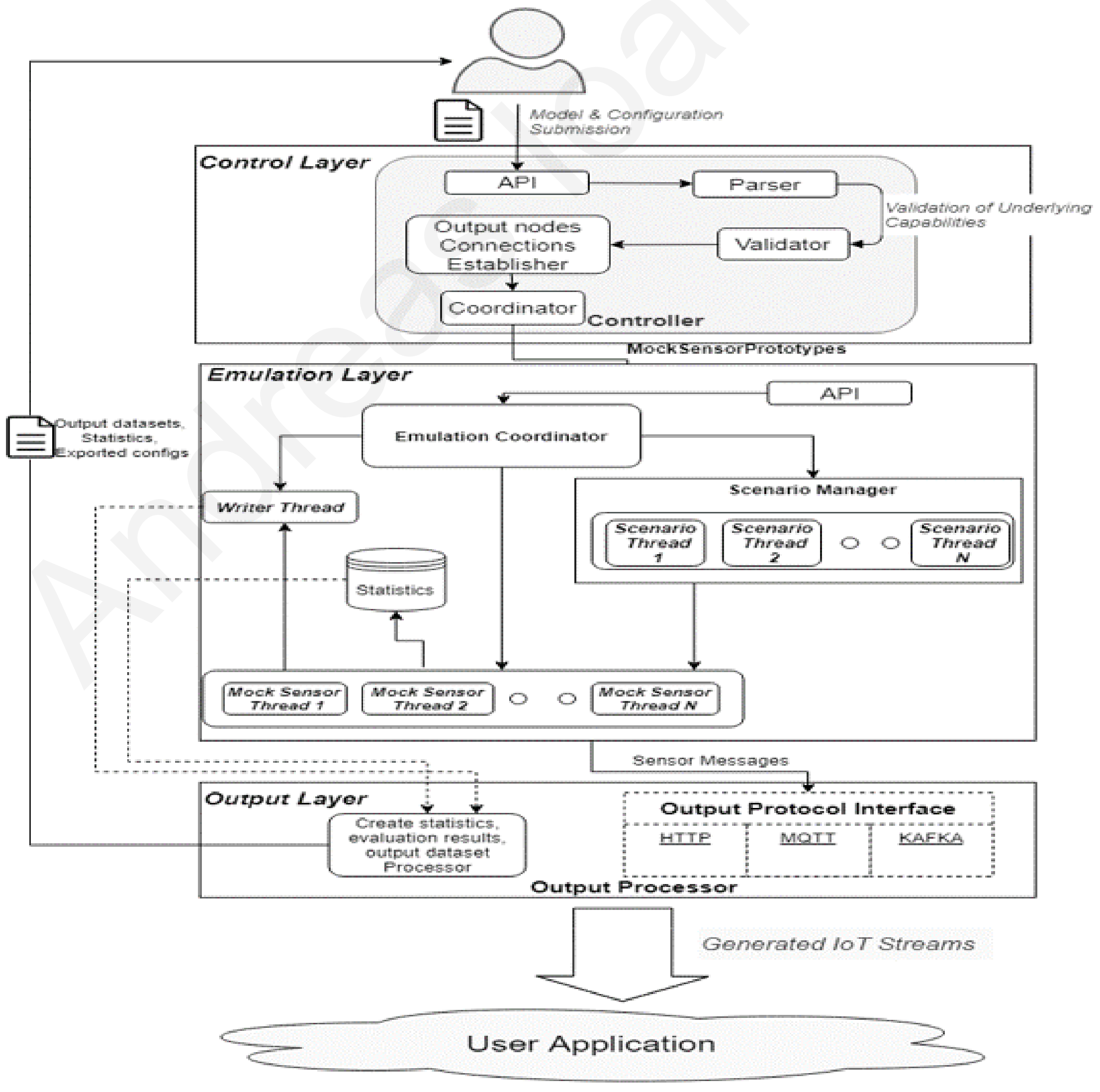


Figure 5.3.1 1: Produce Mock Sensor Data Flow

5.3.2 Class Modelling

One step before we move on to the flow, let me introduce some main concepts and models that will be needed. Currently we are aware of the **MockSensorPrototype** entity. In this implementation, I introduced some other important models as well that affect the understanding of the implementation. To start with, there is the ***MockSensorPrototypeJob*** class, which is actually a wrapper class for MockSensorPrototype, that keeps also some runtime information about the MockSensorPrototype (the current running MockSensors for the relative MockSensorPrototype), and the mapping is one MockSensorPrototypeJob per MockSensorPrototype. The next class that I would like to mention, is the ***MockSensor***. This class is a plain java model, where it could be considered as the representative model for each MockSensor that will be created. Each MockSensor, it is related with its MockSensorPrototype. Then we have the ***MockSensorJob*** class, which is maybe the most important one, because this class represents each **running** MockSensor. It can be considered as the running instance of each MockSensor, since each MockSensorJob has a reference to its MockSensor (that implies also to the MockSensorPrototype). All the functional logic for each MockSensor is located and can be executed in its own corresponding MockSensorJob, since each MockSensorJob it is a different thread, that has purpose to generate messages according to it is reference MockSensorPrototype properties.

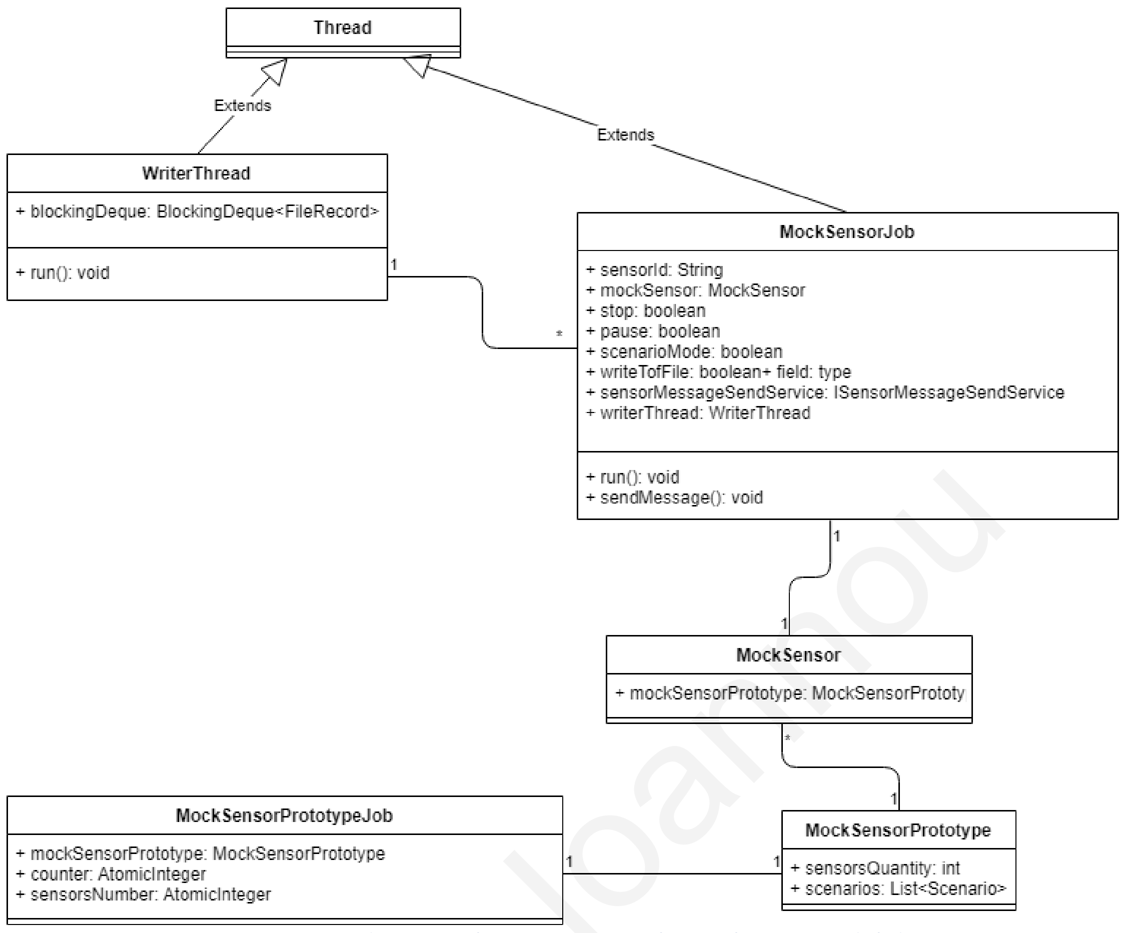

Thread
Extends
Extends
WriterThread
+ blockingDeque: BlockingDeque<FileRecord>
+ run(): void
1
*
MockSensorJob
+ sensorId: String
+ mockSensor: MockSensor
+ stop: boolean
+ pause: boolean
+ scenarioMode: boolean
+ writeTofFile: boolean+ field: type
+ sensorMessageSendService: ISensorMessageSendService
+ writerThread: WriterThread
+ run(): void
+ sendMessage(): void
1
1
MockSensor
+ mockSensorPrototype: MockSensorPrototy
*
1
MockSensorPrototypeJob
+ mockSensorPrototype: MockSensorPrototype
+ counter: AtomicInteger
+ sensorsNumber: AtomicInteger
1
1
MockSensorPrototype
+ sensorsQuantity: int
+ scenarios: List<Scenario>

Figure 5.3.2 1: Produce Mock Sensor Data class relation model diagram

It is easier to visualize the association, for all the previously mentioned classes in the Figure 5.3.2 1 above. We can see that each MockSensorPrototype is associated only with one MockSensorPrototypeJob and each MockSensorPrototypeJob is associated with one MockSensorPrototype, but one MockSensorPrototype can be associated with multiple MockSensors. It is reasonable, since each MockSensorPrototype can have multiple MockSensors, but on the other hand, it is strict that one MockSensor can belong only to one MockSensorPrototype. Moving on, each MockSensor is associated only with one MockSensorJob (and the reverse). The MockSensorJob is extending the ***Thread*** java class, since it is a Thread, and is related with another Thread, the ***WriterThread***, if the user has provided the *outputFile* for the MockSensorPrototype that the MockSensor of the MockSensorJob is associated. It is worth to note here, that all the information that the user has provided in the configs_file.json regarding the mock sensor prototype can be found in the MockSensorPrototype (that will be cast through the input mapping from the configs_file.json) . Beyond that, everything else is created at runtime after processing and validations.

5.3.3 Flow

Having mentioned the above, now we have a basic idea for the main and associated classes that this implementation is consisting of. Now let us go straight to the workflow.

In the

Figure 5.3.3 1, the high-level flow for the *initiate* method of the Produce Mock Sensor Data Implementation can be seen. The whole flow starts after the *Controller* invokes the *initiate* method, that this implementation has overridden of the SensorPrototype Input Interface. Once the method starts, it will attempt to create and process the MockSensorPrototypeJobs, using the MockSensorPrototypes provided by the Controller. In this processing procedure, for each MockSensorPrototype a corresponding MockSensorPrototypeJob will be created and initialized, and then it will be validated if all the required fields are provided by the user. After some processing to cast and transform some of the fields that the user provided into data models, the MockSensorPrototypeJobs will be ready for use. In the continuation, if at least one provided MockSensorPrototype has the field *outputFile* (which means the user wants the data populated for all the sensors for the mock sensor prototype to be exported in a file) a *WriterThread* thread will be created*.* Some processing will happen in order for the WriterThread to store the information of the output file (name and file extension) for all the mock sensor prototypes that their data will be exported.

This WriterThread which will have only one instance, it will have a blocking queue, where the sensor messages that each MockSensorJob will be producing will be stored. The job for the WriterThread, is to pop the messages from its queue and attempt to write them to the according file (each mock sensor prototype can have different output filename). After this procedure, MockSensors will be created (one MockSensor model for every sensor in the MockSensorPrototype *sensorsQuantity* value) and will be linked with its own MockSensorPrototype. All the MockSensors quantity would be equal to the sum of all the sensorsQuantity values for all the MockSensorPrototypes). Now, is time for the real sensor Threads to be created. Based on the MockSensors, for each MockSensor, a MockSensorJob will be created, and will start to generate messages based on the MockSensorPrototype of its referenced MockSensor. A thing to add here, is that the MockSensorJob will have as reference also the one and only WriterThread (if the MockSensorPrototype of its referenced MockSensor was set to export) and the one and only Output Protocol Interface, that the Controller passed to the *initiate* method. At this point every emulated sensor (MockSensorJob) has started to producing data, and passing each produced message to the blockingQueue of the WriterThread, in order to be written to the according file if the user has declared so, and then to the ISensorMessageSendService implementation in the output layer in order to be forwarded to the user's application. Last step before the initiate method for this implementation finishes, is to handle the scenarios. If at least a MockSensorPrototype has at least one scenario, the scenario will be validated, and if it is valid, the ScenarioManager will be initialized and schedule the scenario.

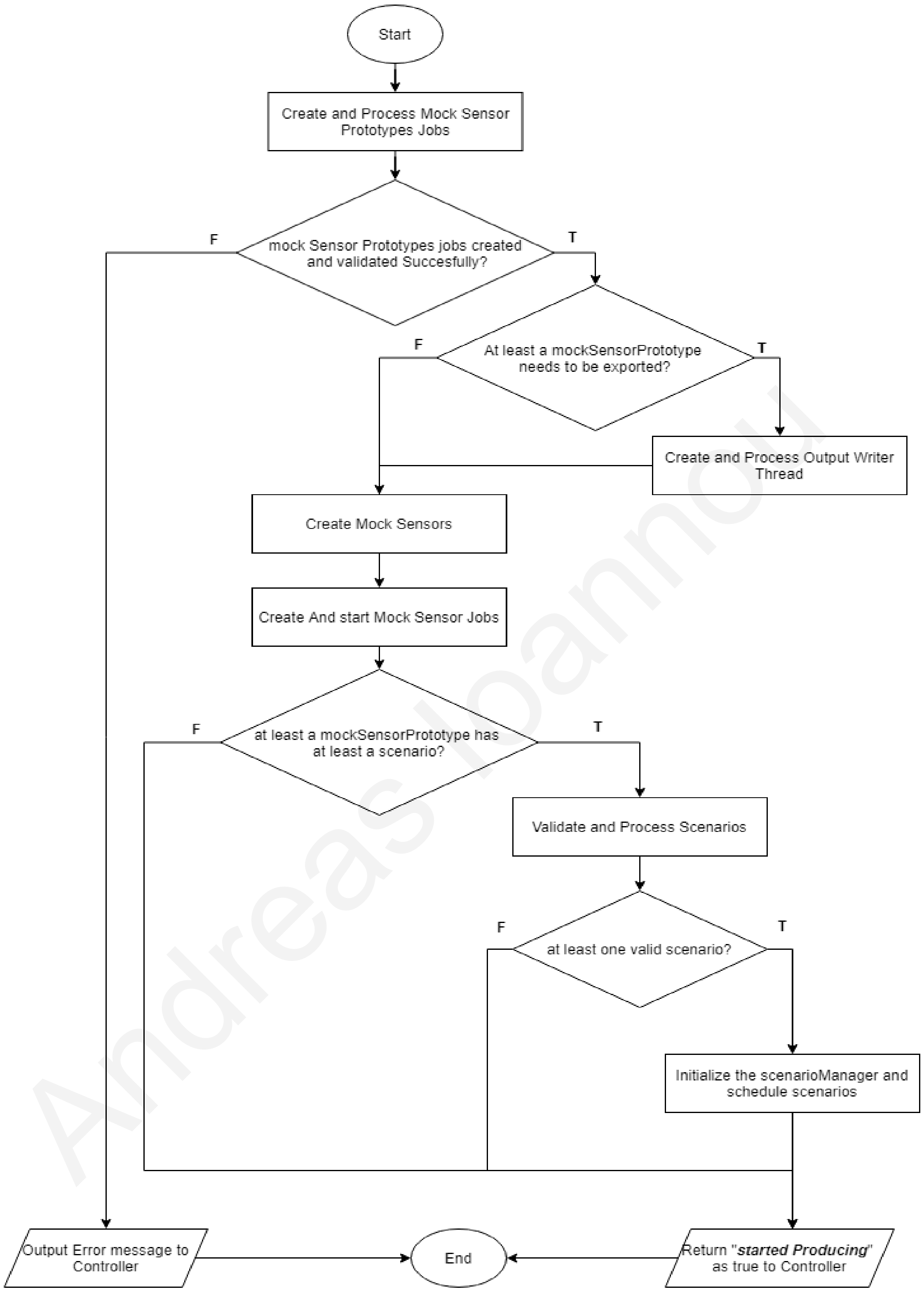

Start
Create and Process Mock Sensor Prototypes Jobs
mock Sensor Prototypes jobs created and validated Succesfully?
F
T
At least a mockSensorPrototype needs to be exported?
F
T
Create and Process Output Writer Thread
Create Mock Sensors
Create And start Mock Sensor Jobs
at least a mockSensorPrototype has at least a scenario?
F
T
Validate and Process Scenarios
at least one valid scenario?
F
T
Initialize the scenarioManager and schedule scenarios
Output Error message to Controller
End
Return "started Producing" as true to Controller

Figure 5.3.3 1: Produce Mock sensor data flow diagram

The scenario management can be seen in the Figure 5.3.3 2 below. Each MockSensorPrototype can have multiple scenarios, and each scenario can be associated only with one sensor, through the "*sensoId*". If at least scenario is found in at least a MockSensorPrototype, the *ScenarioManager* will be initialized, and create one ***ScenarioJob*** thread per ***Scenario***, passing also to the ScenarioJob the MockSensorJob thread that will be informed for the according scenario (Scenario's *sensorId* and MockSensorJob's *sensorId* would be the same). Once the ScenarioManager creates all the needed ScenarioJob threads, it can be idle, until the user terminates or pauses the workload generator. Meanwhile, each created ScenarioJob thread, will have a timer and will be waiting for the amount of seconds provided in "*scenarioDelay"* for the scenario. Once the scenario time comes, the ScenarioJob thread will inform the corresponding MockSensorJob thread, by calling MockSensorJob's *triggerScenario*(Scenario scenario) method and pass the scenario. After that it will be waiting until the *scenarioDuration* time in seconds has passed in order to inform the MockSensorJob that the scenario is over, by calling the *terminateScenario()* method. Finally, the ScenarioJob will terminate smoothly. Meanwhile from the time that the ScenarioJob called the *triggerScenario* method of the MockSensorJob, the MockSensorJob thread will be in *scenarioMode* and will be generating values according to the provided scenario's *scenarioFieldValueInfoList* that stores which fields and what value they will be generating (*sensorFieldScenarioName, sensorFieldScenarioGenerationRate)*, thought the scenario until the ScenarioJob calls the *terminateScenario()* method.

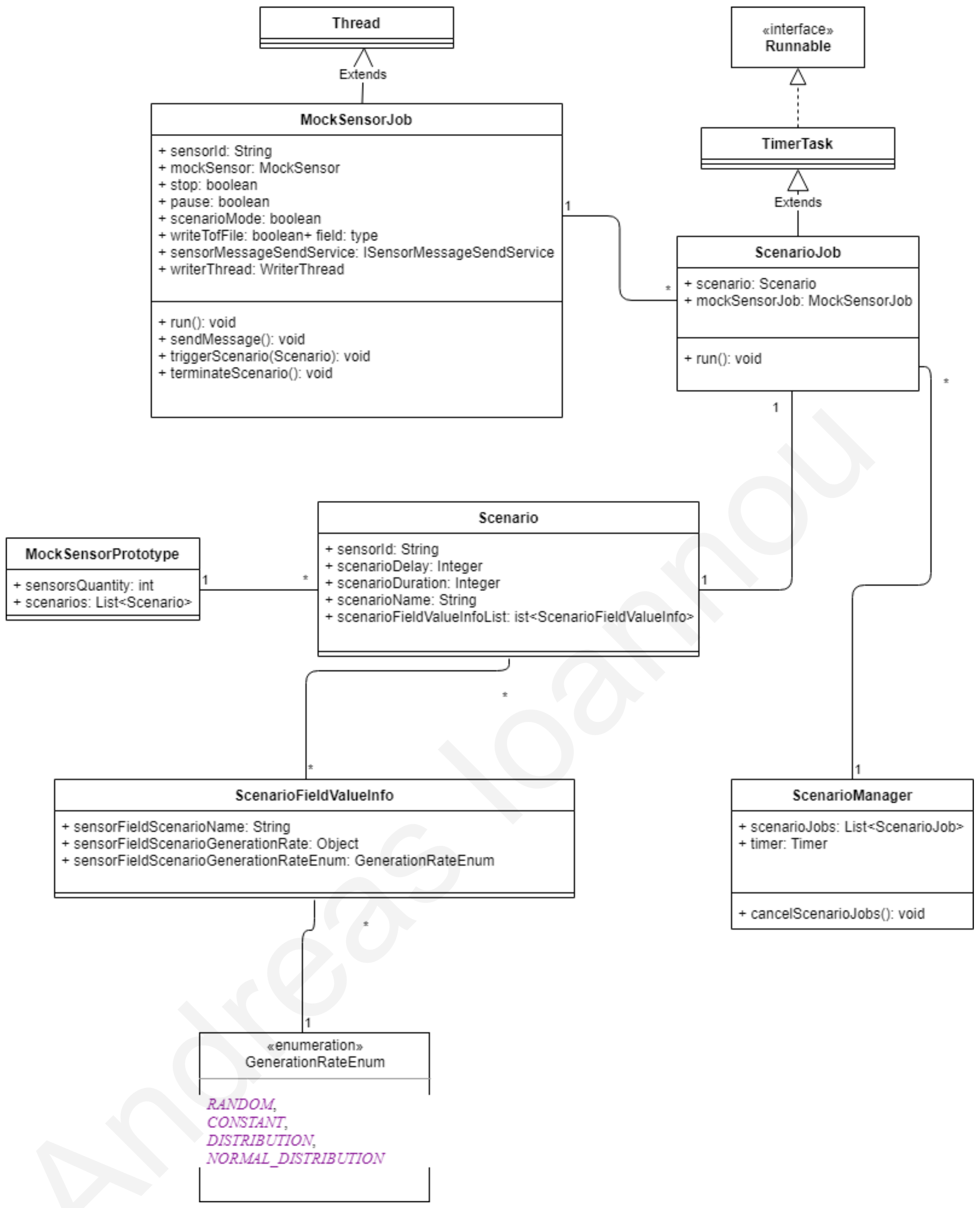

Thread
Extends
MockSensorJob
+ sensorId: String
+ mockSensor: MockSensor
+ stop: boolean
+ pause: boolean
+ scenarioMode: boolean
+ writeTofFile: boolean+ field: type
+ sensorMessageSendService: ISensorMessageSendService
+ writerThread: WriterThread
+ run(): void
+ sendMessage(): void
+ triggerScenario(Scenario): void
+ terminateScenario(): void
«interface»
Runnable
TimerTask
Extends
ScenarioJob
+ scenario: Scenario
+ mockSensorJob: MockSensorJob
+ run(): void
1
*
1
*
MockSensorPrototype
+ sensorsQuantity: int
+ scenarios: List<Scenario>
1
*
Scenario
+ sensorId: String
+ scenarioDelay: Integer
+ scenarioDuration: Integer
+ scenarioName: String
+ scenarioFieldValueInfoList: ist<ScenarioFieldValueInfo>
1
*
*
ScenarioFieldValueInfo
+ sensorFieldScenarioName: String
+ sensorFieldScenarioGenerationRate: Object
+ sensorFieldScenarioGenerationRateEnum: GenerationRateEnum
1
ScenarioManager
+ scenarioJobs: List<ScenarioJob>
+ timer: Timer
+ cancelScenarioJobs(): void
*
1
«enumeration»
GenerationRateEnum
RANDOM,
CONSTANT,
DISTRIBUTION,
NORMAL_DISTRIBUTION

Figure 5.3.3 2: Scenario Management Class Diagram.

The *pause* method implementation is simpler. The state of the workload generator before the method been called from the Controller, is that we have now the sensor threads running producing mock sensor data, and potentially the WriterThread, along with the ScenarioManager and its ScenarioJob threads doing their tasks. Like it is being explained in the Stop action, the purpose of this method is to temporarily pause all the running threads and actions. So, the implementation, firstly cancels all the scheduled or running scenarios (threads) through the ScenarioManager (if it is initialized), and then pauses all the sensor threads. Finally, it pauses also the WriterThread if it is initialized. By now, the sensor threads are in sleep mode, and will be notified to resume their task (to produce mock sensor data again) when and if the user triggers the start action.

Having said that, once the user calls the *start* action (after the *stop* action), it means that he wants to resume (*resume* method will be invoked) the generation of mock data. Then the implementation will trigger the sleeping sensor threads to wake up, by notifying them to resume their data production with the same behaviour. This will happen also to the WriterThread, which if it was sleeping, it will also wake up and start writing sensor messages from its queue to the appropriate output files. An important thing to note, is that the ScenarioManager, and all the previously scheduled scenarios, **will not be rescheduled**, since the scheduled time was with perspective the initialization of the workload generator.

Closing this implementation explanation, we have to also mention the terminate interface method implementation. The *terminate* method which can be called only when the user calls the *restart* action (after the *stop* action), implies that the previous implementation method called was the *pause*, which means that all the threads are currently sleeping. So, the next step is to wake them up and terminate them all smoothly. Precisely, it will terminate all sensor threads and the writer thread, and finally will free the memory for all the mockSensorPrototypes.

5.4 Replay sensor data from Dataset Implementation

In this section, some technical details, the flow and the implementation for the Replay Sensor Data from CSV file will be explained.

5.4.1 Components and Association

Before we go deeper with the implementation, let us transform the high-level workload generator flow from

Figure 3.2.2 *1* by replacing the emulation layer, with the current emulation instance implementation.

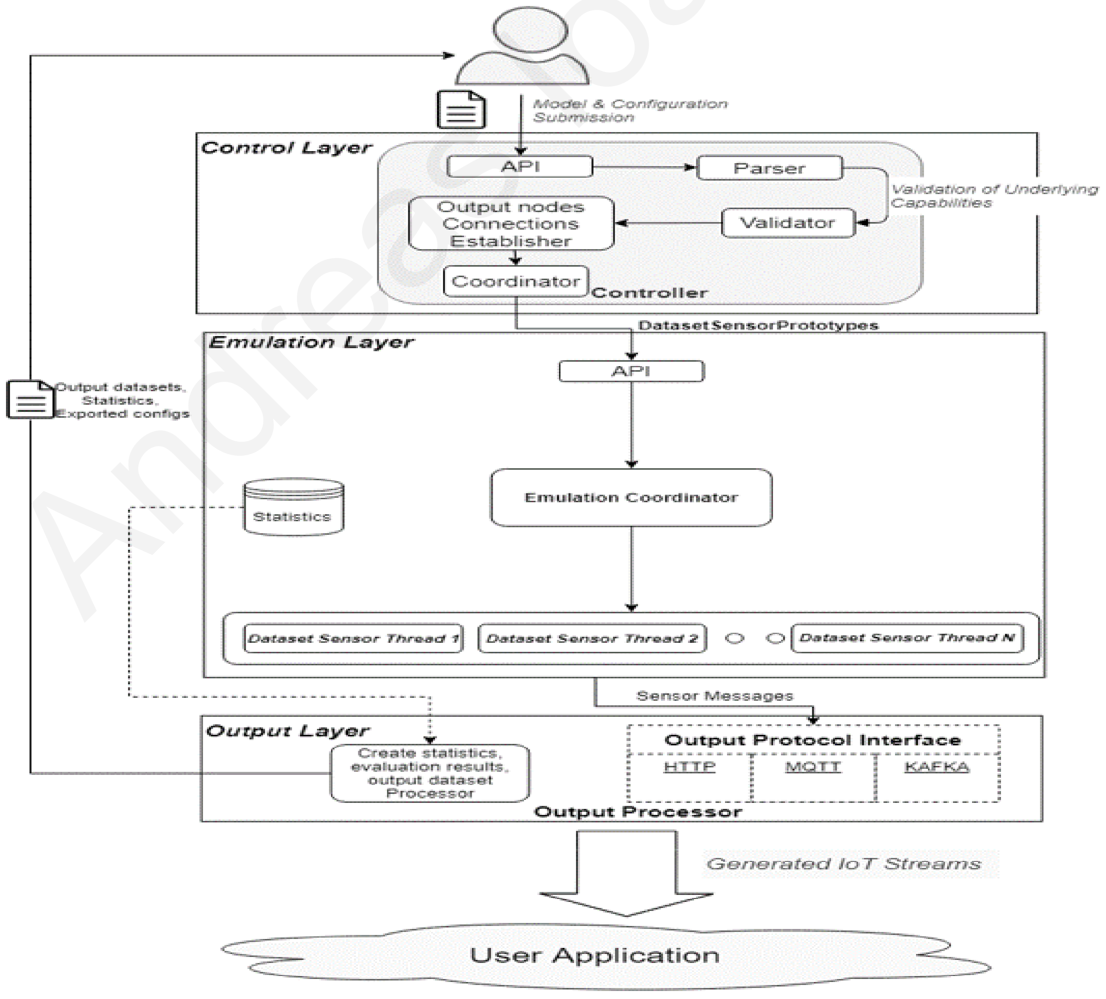


Figure 5.4.1 1: Replay Sensor Data from Dataset Flow

In the Figure 5.4.1 1 above we can see the workload generator flow, in the case that in the emulation layer, we will have the Replay Sensor Data from Dataset emulation instance. The user will provide the *configs_file.json* with the datasetSensorPrototypes, and the controller will do all the processing as the explained in

Figure 3.2.2 *1*, and pass to this emulation instance implementation, the datasetSensorPrototypes along with the desired and initialized output protocol interface implementation. In the emulation layer for this implementation the emulation coordinator handles and monitors the threads for each sensor (for each dataset). Each thread represents a datasetSensorPrototype, because like it is being explained in previous chapters, for this implementation the user provides the dataset that has different exported messages for the same type of sensors. This implementation does not have as many components like the previous one. The important components in this emulation layer are the threads, where each one reproduces the messages accordingly from its dataset, and passes the generated data stream to the output layer in order to be transferred to the user application, and at the same time the threads store some statistics for the output data, for post analysis.

5.4.2 Dataset Thread Modelling

Before moving on to the flow, let us quickly have an overview for the main component for this implementation, which is the thread per dataset that handles internally the reproduction of the dataset, customized.

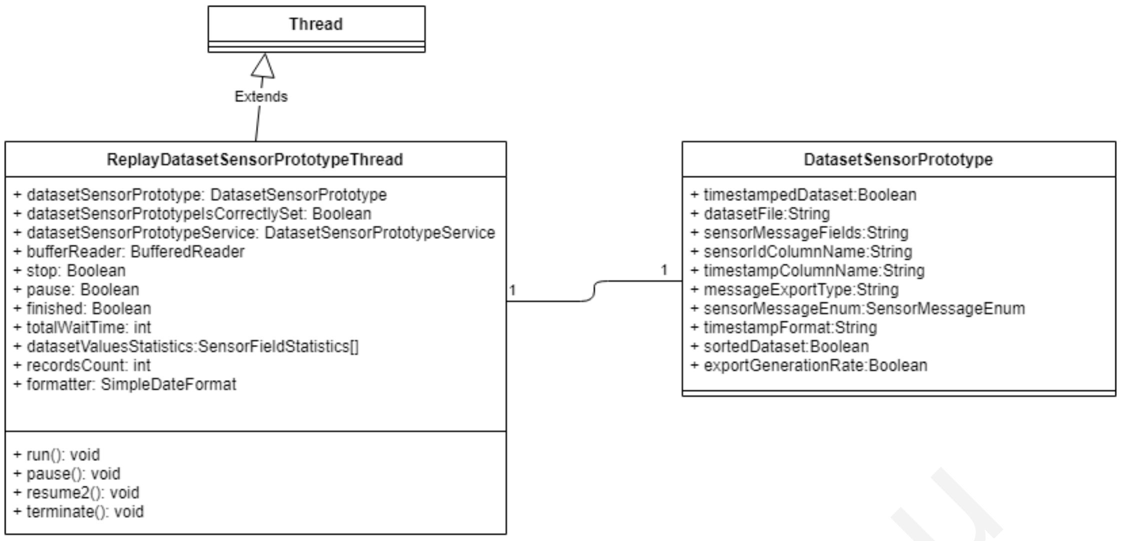

Thread
Extends
ReplayDatasetSensorPrototypeThread
+ datasetSensorPrototype: DatasetSensorPrototype
+ datasetSensorPrototypeIsCorrectlySet: Boolean
+ datasetSensorPrototypeService: DatasetSensorPrototypeService
+ bufferReader: BufferedReader
+ stop: Boolean
+ pause: Boolean
+ finished: Boolean
+ totalWaitTime: int
+ datasetValuesStatistics:SensorFieldStatistics[]
+ recordsCount: int
+ formatter: SimpleDateFormat
+ run(): void
+ pause(): void
+ resume2(): void
+ terminate(): void
1
1
DatasetSensorPrototype
+ timestampedDataset:Boolean
+ datasetFile:String
+ sensorMessageFields:String
+ sensorIdColumnName:String
+ timestampColumnName:String
+ messageExportType:String
+ sensorMessageEnum:SensorMessageEnum
+ timestampFormat:String
+ sortedDataset:Boolean
+ exportGenerationRate:Boolean

Figure 5.4.2 1: Replay Sensor Data from Dataset Main Thread Class Diagram

In the Figure 5.4.2 1 we can see the class diagram and the attributes for the main component for this implementation, the ***ReplayDatasetSensorPrototypeThread***. As it is easy to guess, it extends the Thread java class, and the whole production for the messages happens in the *run* overridden method. Beside the flag attributes that represent its state (*stop*, *pause*, *finished*) it has also some other attributes that handle the statistics and evaluation for the dataset (*recordCount*, *datasetValueStatistics*, *totalWaitTime*). The most important field is of course the *datasetSensorPrototype*, that the thread will have as reference in order to replay the dataset for this SensorPrototype. Some other attributes, are for reading the dataset and parsing and reading the rows (*formatter*, *bufferReader*), and lastly the rest attributes are related with the functional part (*datasetSensorPrototypeService)* that handles the processing, the validations and the passing of the data to the output layer, and the *datasetSensorPrototypeIsCorrectlySet* to confirm that the dataset is valid.

5.4.3 Flow

Now that we have a basic understanding of the main components let us move straight to the implementation, by introducing the flow. Inside the *initialize* method for this implementation, the emulation coordinator creates one *ReplayDatasetSensorPrototypeThread* per DatasetSensorPrototype provided in the datasetSensorPrototype parameter from the **Controller**. Each thread then starts to process its DatasetSensorPrototype and validates if all the required and needed fields are provided and that they are correctly set up and valid. After the success validation and processing of the provided DatasetSensorPrototype, each thread will validate the data file. More precisely, it will make sure that the file exists and can be parsed, and that all the columns needed are there. If all the above procedures succeeded, then the thread will inform the coordinator that it is ready to start, and the coordinator will start it, in order to start replaying its dataset and reproducing the records for the sensors of the dataset. If at least one thread has been processed and validated successfully, the implementation is considered started, and will inform the controller with the *isStartedProducing* method.

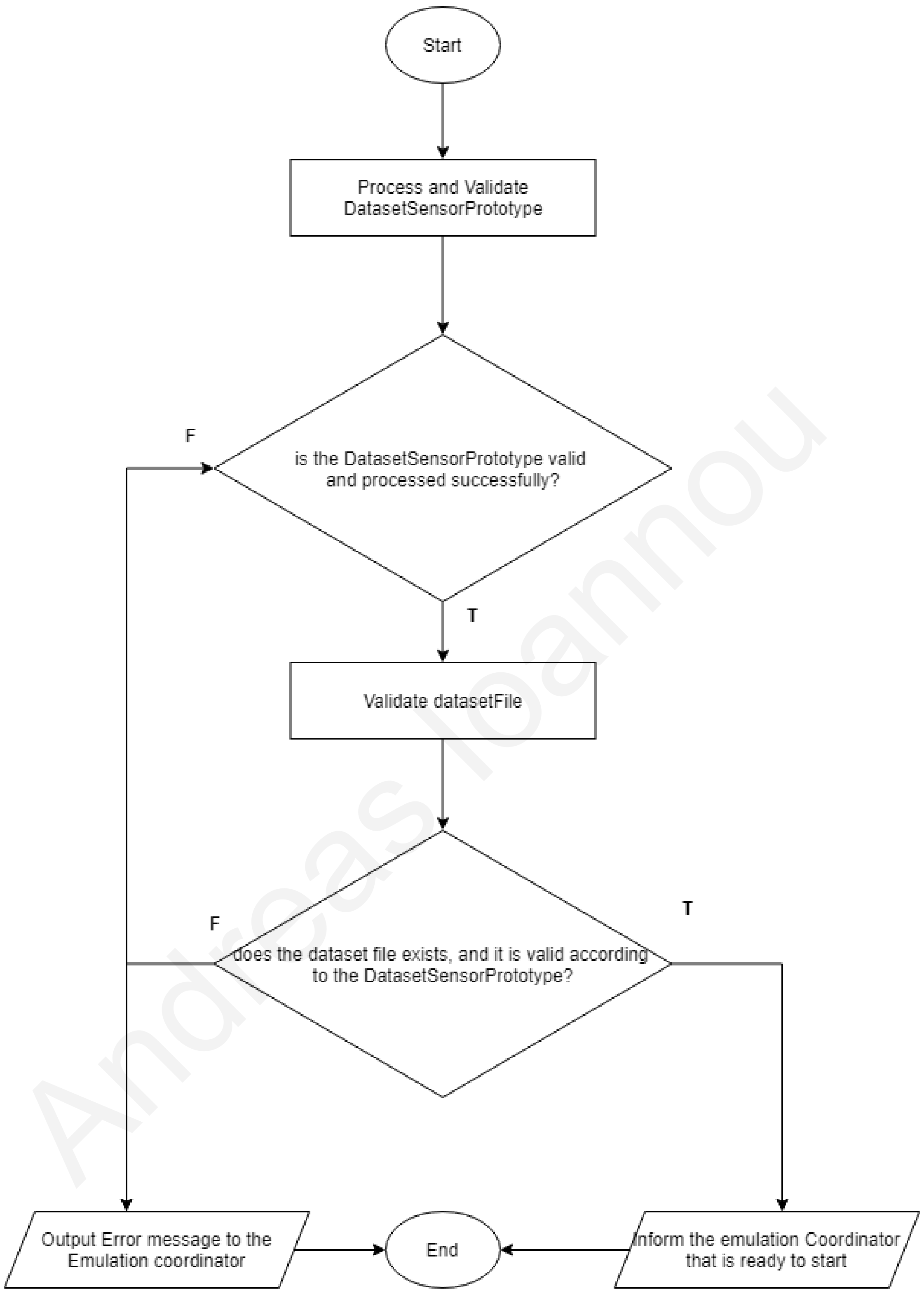

Start
Process and Validate DatasetSensorPrototype
is the DatasetSensorPrototype valid and processed successfully?
F
T
Validate datasetFile
does the dataset file exists, and it is valid according to the DatasetSensorPrototype?
F
T
Output Error message to the Emulation coordinator
End
Inform the emulation Coordinator that is ready to start

Figure 5.4.3 1: Replay Sensor Data from Dataset Thread flow diagram

Figure 5.4.3 1 depicts the flow for each ReplayDatasetSensorPrototypeThread. For the processing and the validation of the DatasetSensorPrototype, it is been validated that either a generation rate or the timestamp information is provided. Then, while validating and processing the dataset file, it verifies that the dataset file is indeed a CSV extension file, that the user provided columns that exist in the dataset (sensorId column and optionally the timestamp column) and finally, it can be sorted if the user declared so (based on the timestamp column for each row). After all the processing and validations happen, the thread starts to reproduce the data from the dataset (according to each row timestamp in comparison to the previous or if the user has provided a generation rate), and it will finish once all the dataset rows have been replayed.

For the *pause*, *resume* and *terminate* methods, the processes are straightforward. In the pause method, the coordinator pauses all the running threads (those that are still replaying rows), where in the resume, the coordinator will trigger the paused threads to resume the reproduction, from where they left. Finally, in the terminate method, the coordinator will notify all the running threads to stop replaying, and finish.

5.5 Output protocol Implementations

In this subchapter, we will briefly say some things for the implementations for the output protocol interface. As explained previously, the currently supported output protocols are HTTP, MQTT, and using KAFKA broker server.

For the HTTP implementation, the validations happening in the *validateAndProcessConfigs* implemented method, is to verify that the user has provided the *requestURI* inside the *protocolConfigs* that the sensor messages will be sent to, along with the *httpServers* information. In the *initializeConnections* and *terminate* methods, the behaviour for this implementation is to do nothing, since the library used for HTTP calls does not need to establish any connection or terminate in order to send HTTP messages. The library used is the *RestTemplate* [5] Class of the Spring Framework [6]. Using the provided methods of this class we are able to send HTTP requests (of all methods) to the output servers within the *sendMessage* method along with the needed request headers (the contentType that states the message format e.g.: JSON, XML, TEXT).

For the other 2 implementations for the pub/sub protocol, MQTT and KAFKA), the behaviour is the same, with the only difference in the library used. The MQTT implementation is using the *Eclipse Paho* [7] library to connect and send messages to the MQTT brokers, and the KAFKA implementation is using *Apache Kafka* [8] java library to connect and send the messages to a KAFKA broker. Having said that, in the *validateAndProcessConfigs* method, in both the implementations it is being checked, whether the user has provided the desired *topic*. On top of that, the *mqttBrokers* field is mandatory for the MQTT implementation and the *kafkaBrokerClusters* for the KAFKA respectively. In the *initializeConnections* both of the implementations are setting the properties for the Client Producer based on the library they are using, in order to create Producer Clients, to send messages using the corresponding libraries

[5] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html
[6] https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html
[7] https://www.eclipse.org/paho/
[8] https://kafka.apache.org/documentation/

method through the *sendMessage* method to the desired brokers. Finally, they both disconnect and close their Producer Clients inside the *terminate* method.

5.6 API Endpoints

In this last sub chapter of the **Implementation** chapter, we will provide the API endpoints that the user can use to trigger actions or to get information for the framework emulation instance. It is worth noting that the APIs are implemented based on the Spring Boot framework [9] architecture, which runs on top of Apache Tomcat servlet Container [10]. The deployed application container will be listening to the same configurable port for all of the APIs.

5.6.1 Workload Generator Actions APIs

The following APIs can be used to trigger workload Generator actions.

POST ***/workloadGenerator/start***

Description	The user can use this API to trigger the execution of the application. This API is the entry point of data generation, and translated to the *start* user action explained in section 5.1.1
PATH	***/workloadGenerator/start***
Method	**POST**
Payload	The payload of the API is an empty JSON object: {}
Query Params	The API can support one query param, called ***delay*** (in seconds) , which like the name implies, is the desired delay until the workload generator starts to produce data.

[9] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
[10] http://tomcat.apache.org/tomcat-8.0-doc/

Response:	Success message (200 HTTP Response Code) if at least one SensorPrototype input implementation was successful, else detailed error message for some validation or missing input for the user (400 HTTP Response Code) or (500 HTTP Response Code) for any unexpected error.

POST ***/workloadGenerator/stop***

Description	The user can use this API to trigger the pause or stop of the application. This API can be used only after the **POST** ***/workloadGenerator/start*** API and is translated to the *stop* user action explained in section 5.1.2 which pauses the data generation and all the pending processes, along with the workload generator.
PATH	***/workloadGenerator/stop***
Method	***POST***
Payload	the payload of the API is an empty JSON object: {}
Query Params	The API can support one query param, called ***delay*** (in seconds) , which like the name implies, is the desired delay until the workload generator stops data production.
Response	Success message (200 HTTP Response Code) if the workload generator was stopped successfully, else detailed error message for some validation error when calling the API (400 HTTP Response Code) or (500 HTTP Response Code) for any unexpected error.

POST ***/workloadGenerator/restart***

Description	The user can use this API to make the workload generator terminate all the current idle sensors and tasks, and to create new sensors based on the new input provided in the *configs_file.json*. Its main purposes are for the user to change input for testing different scenarios. This API can be used only after the **POST** ***/workloadGenerator/stop API*** and is translated to the *restart* user action explained in the section 5.1.3
PATH	***/workloadGenerator/restart***
Method	***POST***
Payload	The payload of the API is an empty JSON object: {}

Query Params	The API can support one query param, called ***delay*** (in seconds) , which like the name implies, is the desired delay until the workload generator restarts again the initialization flow with the new input.
Response	Success message (200 HTTP Response Code) if the workload generator has restarted successfully, else detailed error message for some validation error when calling the API or wrong input (400 HTTP Response Code) or (500 HTTP Response Code) for any unexpected error.

5.6.2 Sensor Actions APIs

The following APIs can be used to retrieve, create new or delete mockSensors and mockSensorPrototypes on runtime when the workload Generator is running and mockSensorPrototype input is initialized. The APIs are supporting only the mockSensorPrototype implementation.

GET */mockSensors*

Description	The user can use this API to get the total count and a list for all the created mockSensors along with each mockSensor's mockSensorPrototype.
PATH	***/mockSensors***
Method	***GET***
Payload	-
Query Params	The API can support one query param, called ***sensorId***, which is to filter the response only for the specified sensor.
Response	Total count and list of all the mockSensors information, or information only for the sensor having the specified query (200 HTTP Response Code). Bad Request error If the workloadGenerator is not running (400 HTTP Response Code) or no mockSensorPrototype is initialized , Not Found (404 HTTP Response Code) if the mockSensor with specified sensorId does not exists, (500 HTTP Response Code) for any unexpected error.

POST */mockSensors*

Description	The user can use this API to create and add more mockSensors at runtime for an already existing mockSensorPrototype.
PATH	***/mockSensors***
Method	***POST***

Payload	The payload of the API is a JSON object with the needed quantity and the desired mockSensorPrototype' Name:
	`{`
	`    "quantity": 15,`
	`    "mockSensorPrototypeName": "temperature_sensor"`
	`}`

Query Params	-
Response	Success Created response (201 HTTP Response Code) which implies that the mockSensors were created successfully, or Bad Request error message (400 HTTP Response Code) if some parameter in the payload is not provided or no mockSensorPrototype is initialized, Not Found (404 HTTP Response Code) if the mockSensorPrototypeName does not exists and (500 HTTP Response Code) for any unexpected error.

DELETE */mockSensors/{mockSensorId}*

Description	The user can use this API to stop the production and delete a mockSensor
PATH	***/mockSensors/{mockSensorId}***
Method	***DELETE***
Payload	-
Path Params	The API requires one path param, called ***mockSensorId***, which is the Id of the mockSensor to be deleted
Response	Success No Content response (204 HTTP Response Code) which implies that the mockSensor specified was deleted, or Not Found (404 HTTP Response Code) if the mockSensorId provided does not exist, (500 HTTP Response Code) for any unexpected error.

GET ***/mockSensorPrototypes***

Description	The user can use this API to get information about mockSensorPrototypes
PATH	***/mockSensorPrototypes***
Method	***GET***
Payload	-
Query Params	The API can support one query param, called ***mockSensorPrototypeName***, which is to filter the response only for the specified mockSensorPrototype.
Response	Total count and list of all the mockSensorPrototypes information , or information only for the mockSensorPrototype having the specified query param as name (200 HTTP Response Code). Bad Request error If the workloadGenerator is not running or no mockSensorPrototype is initialized (400 HTTP Response Code), Not Found (404 HTTP Response Code) if the mockSensorPrototype with specified name does not exists, (500 HTTP Response Code) for any unexpected error.

POST ***/mockSensorPrototypes***

Description	The user can use this API to create new mockSensorPrototypes at runtime
PATH	***/mockSensorPrototypes***
Method	***POST***
Payload	The payload of the API is JSON object same as the input for the mockSensorPrototype mapping in the configs_file.json
Query Params	-
Response	Success Created response (201 HTTP Response Code) which implies that the mockSensorPrototype and its mockSensors were created successfully and started generating data successfully, or Bad Request error message (400 HTTP Response Code) if some mandatory parameter in the payload is not provided, and (500 HTTP Response Code) for any unexpected error.

DELETE ***/mockSensorPrototypes/{mockSensorPrototypeName}***

Description	The user can use this API to stop the production and delete all the sensors for the specified mockSensorPrototype and then delete it
PATH	***/mockSensorPrototypes/{mockSensorPrototypeName}***
Method	***DELETE***
Payload	-
Path Params	The API requires one path param, called ***mockSensorPrototypeName***, which is the name of the mockSensorPrototype to be deleted
Response	Success No Content response (204 HTTP Response Code) which implies that the mockSensorPrototype specified was deleted, or Not Found (404 HTTP Response Code) if the mockSensorPrototypeName provided does not exist and (500 HTTP Response Code) for any unexpected error.

Chapter 6

System Evaluation and Results

6.1 Usability Evaluation

6.1.1 Heterogeneity and Sensor Model Realizaiton Evaluation

6.1.2 Accuracy Evaluation

6.1.3 Extensibility Configurability Customization Evaluation

6.1.4 Usability evaluation, through Integration with real Fog Emulator on a real scenario use case

6.1.4.1 Increasing Workload Test

6.1.4.2 Data Skewness Experiment

6.2 Performance Evaluation and limitations

6.2.1 Sensor Message Delay

6.2.1.1 Input and Evaluation Methodology

6.2.1.2 Evaluation Experiments

6.2.1.2.1 Sensor Quantity Factor

6.2.1.2.2 Generation Rate Factor

6.2.1.2.3 Host Machine Resources Factor

6.2.1.2.4 Sensors Quantity Limitation

6.3 Evaluation Results Summary and System limitations

In this chapter, the implemented system will be evaluated, based on the initial requirements we set back in section 3.1. In addition, we will show its usability with some real scenarios in combination with a fog emulator, then we will do some experiments to find what factors influence the performance, and finally we will find the limits of our system.

6.1 Usability Evaluation

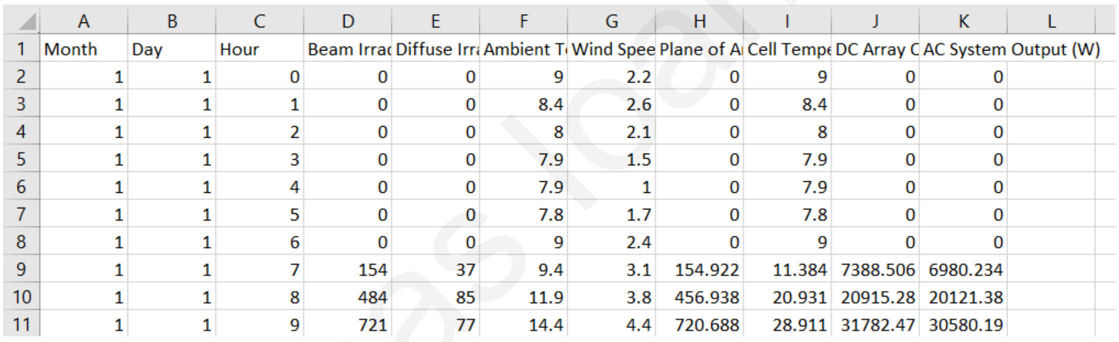
In order to evaluate the system, let us first recall the requirements that such a system should fulfil. When we were introducing the idea for a workload generator for IoT data, back to section 3.1, we mentioned some pillars that the system should support in order to be a workload generator for IoT devices that any user could use, and users could actually be benefited by using it. In this section we will try to demonstrate the heterogeneity and the accuracy requirements.

6.1.1 Heterogeneity and Sensor Model Realization Evaluation

One of the most important challenges and requirements for a workload generator stated, was the Heterogeneity that the system should support, based on the wide variety of different IoT devices, sensors, their message formats, the output protocol and generally the range and differences between the scenarios that the users would like to test.

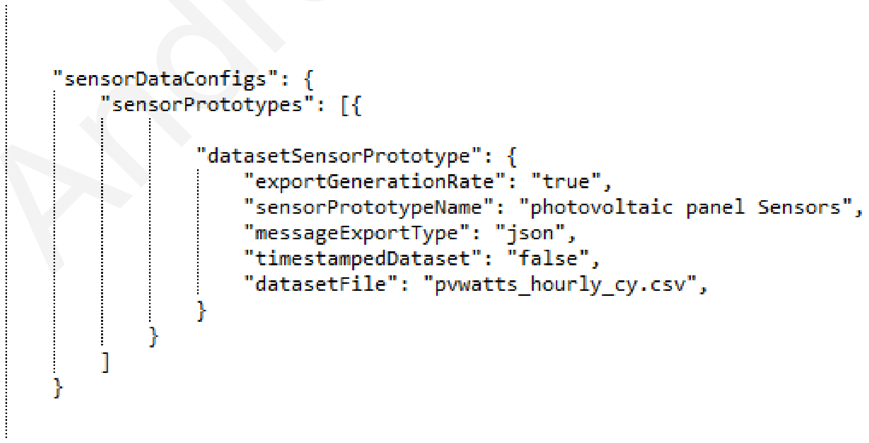
Our system can support currently 3 output protocols, and can be extended to support even more, and in addition it supports 3 of the most used message formats (JSON, XML, TEXT) , which also can be extended for more. On top of that, a wide range of sensors, and their properties can be achieved, using our abstract realization model. The system gives the ability to the user to provide any message field, or even nested fields, that results in a flexible usage in order to adjust the real sensors with the system model mapping. Last but not least, based on the abstraction and the correct realization of our sensor model for the implementation of replaying real data, the user has the ability to provide almost any dataset, with real exported sensor data, in order to be replayed, boosting the heterogeneity and the generality of our realization models.

Let us demonstrate some of the above flexibilities, with some file instances for different CSV datasets for completely different applications (and different sensor types) and their corresponding input configs_file.json mapping configurations.



	A	B	C	D	E	F	G	H	I	J	K	L
1	Month	Day	Hour	Beam Irrac	Diffuse Irra	Ambient T	Wind Spee	Plane of A	Cell Tempe	DC Array C	AC System Output (W)	
2	1	1	0	0	0	9	2.2	0	9	0	0	
3	1	1	1	0	0	8.4	2.6	0	8.4	0	0	
4	1	1	2	0	0	8	2.1	0	8	0	0	
5	1	1	3	0	0	7.9	1.5	0	7.9	0	0	
6	1	1	4	0	0	7.9	1	0	7.9	0	0	
7	1	1	5	0	0	7.8	1.7	0	7.8	0	0	
8	1	1	6	0	0	9	2.4	0	9	0	0	
9	1	1	7	154	37	9.4	3.1	154.922	11.384	7388.506	6980.234	
10	1	1	8	484	85	11.9	3.8	456.938	20.931	20915.28	20121.38	
11	1	1	9	721	77	14.4	4.4	720.688	28.911	31782.47	30580.19	

Figure 6.1.1.1: Screenshot from a dataset that measured annual photovoltaic panels data in Cyprus [11]



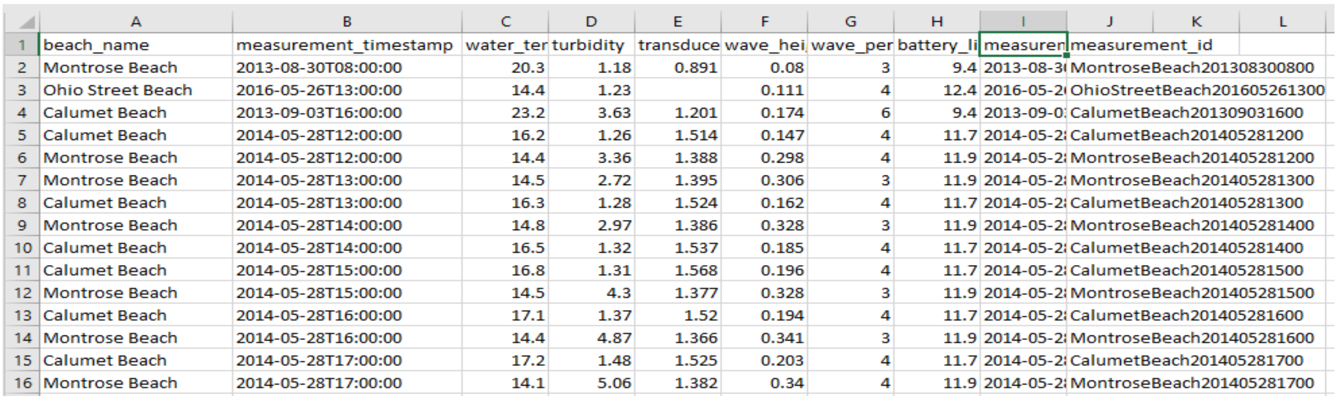
```
"sensorDataConfigs": {
    "sensorPrototypes": [{

            "datasetSensorPrototype": {
                "exportGenerationRate": "true",
                "sensorPrototypeName": "photovoltaic panel Sensors",
                "messageExportType": "json",
                "timestampedDataset": "false",
                "datasetFile": "pvwatts_hourly_cy.csv",
            }
        }
    ]
}
```

Figure 6.1.1.2: Sensor Modelling Input inside configs_file.json for dataset from

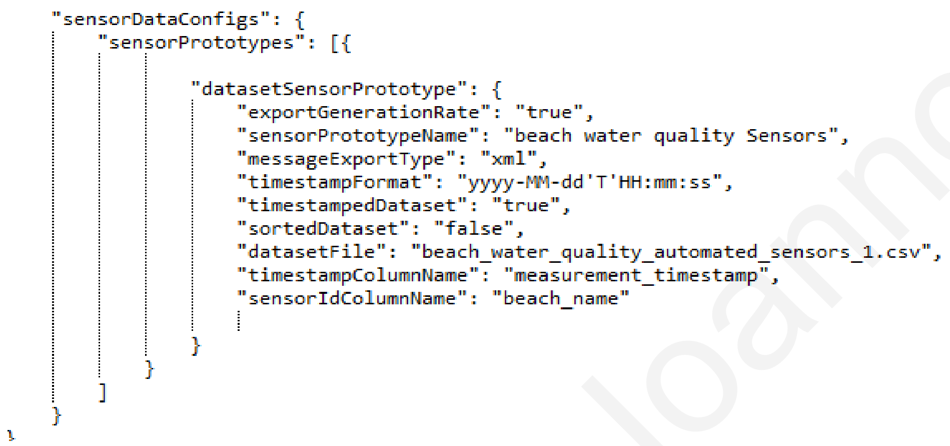
Figure 6.1.1.1

[11] https://enedi.eu/



	A	B	C	D	E	F	G	H	I	J	K	L
1	beach_name	measurement_timestamp	water_ter	turbidity	transduce	wave_hei	wave_per	battery_li	measuren	measurement_id		
2	Montrose Beach	2013-08-30T08:00:00	20.3	1.18	0.891	0.08	3	9.4	2013-08-3(	MontroseBeach201308300800		
3	Ohio Street Beach	2016-05-26T13:00:00	14.4	1.23		0.111	4	12.4	2016-05-2(	OhioStreetBeach201605261300		
4	Calumet Beach	2013-09-03T16:00:00	23.2	3.63	1.201	0.174	6	9.4	2013-09-0:	CalumetBeach201309031600		
5	Calumet Beach	2014-05-28T12:00:00	16.2	1.26	1.514	0.147	4	11.7	2014-05-2:	CalumetBeach201405281200		
6	Montrose Beach	2014-05-28T12:00:00	14.4	3.36	1.388	0.298	4	11.9	2014-05-2:	MontroseBeach201405281200		
7	Montrose Beach	2014-05-28T13:00:00	14.5	2.72	1.395	0.306	3	11.9	2014-05-2:	MontroseBeach201405281300		
8	Calumet Beach	2014-05-28T13:00:00	16.3	1.28	1.524	0.162	4	11.7	2014-05-2:	CalumetBeach201405281300		
9	Montrose Beach	2014-05-28T14:00:00	14.8	2.97	1.386	0.328	3	11.9	2014-05-2:	MontroseBeach201405281400		
10	Calumet Beach	2014-05-28T14:00:00	16.5	1.32	1.537	0.185	4	11.7	2014-05-2:	CalumetBeach201405281400		
11	Calumet Beach	2014-05-28T15:00:00	16.8	1.31	1.568	0.196	4	11.7	2014-05-2:	CalumetBeach201405281500		
12	Montrose Beach	2014-05-28T15:00:00	14.5	4.3	1.377	0.328	3	11.9	2014-05-2:	MontroseBeach201405281500		
13	Calumet Beach	2014-05-28T16:00:00	17.1	1.37	1.52	0.194	4	11.7	2014-05-2:	CalumetBeach201405281600		
14	Montrose Beach	2014-05-28T16:00:00	14.4	4.87	1.366	0.341	3	11.9	2014-05-2:	MontroseBeach201405281600		
15	Calumet Beach	2014-05-28T17:00:00	17.2	1.48	1.525	0.203	4	11.7	2014-05-2:	CalumetBeach201405281700		
16	Montrose Beach	2014-05-28T17:00:00	14.1	5.06	1.382	0.34	4	11.9	2014-05-2:	MontroseBeach201405281700		

Figure 6.1.1.3: Screenshot of a dataset that measured beach water quality in Chicago [12]

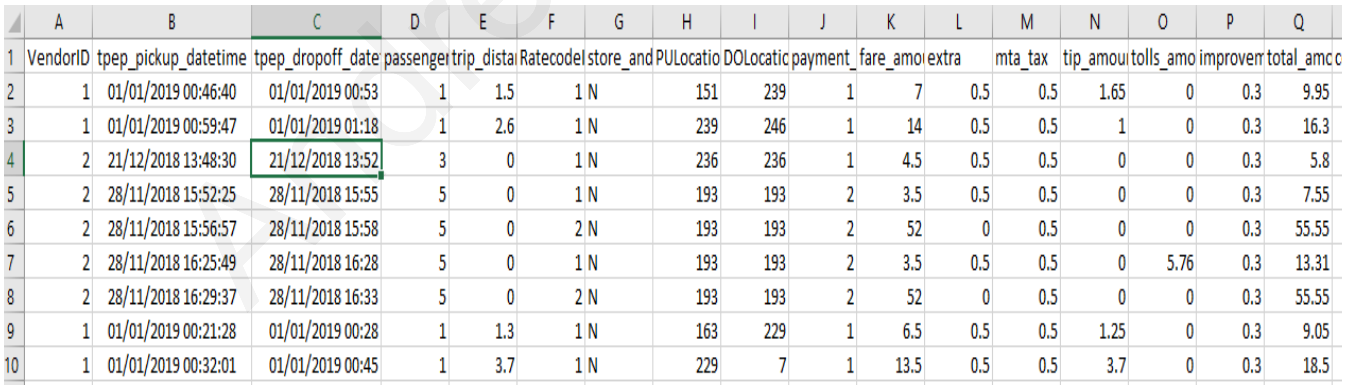


```
"sensorDataConfigs": {
    "sensorPrototypes": [{

            "datasetSensorPrototype": {
                "exportGenerationRate": "true",
                "sensorPrototypeName": "beach water quality Sensors",
                "messageExportType": "xml",
                "timestampFormat": "yyyy-MM-dd'T'HH:mm:ss",
                "timestampedDataset": "true",
                "sortedDataset": "false",
                "datasetFile": "beach_water_quality_automated_sensors_1.csv",
                "timestampColumnName": "measurement_timestamp",
                "sensorIdColumnName": "beach_name"

            }
        }
    ]
}
```

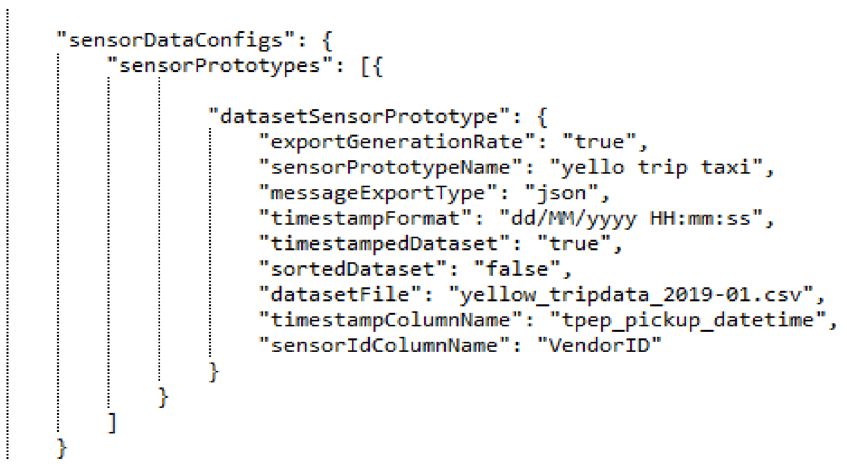
Figure 6.1.1.4:Sensor Modelling Input inside configs_file.json for dataset from Figure 6.1.1.3



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	VendorID	tpep_pickup_datetime	tpep_dropoff_date	passenger	trip_dista	Ratecodel	store_and	PULocatio	DOLocatic	payment_	fare_amo	extra	mta_tax	tip_amou	tolls_amo	improvem	total_amc
2	1	01/01/2019 00:46:40	01/01/2019 00:53	1	1.5	1	N	151	239	1	7	0.5	0.5	1.65	0	0.3	9.95
3	1	01/01/2019 00:59:47	01/01/2019 01:18	1	2.6	1	N	239	246	1	14	0.5	0.5	1	0	0.3	16.3
4	2	21/12/2018 13:48:30	21/12/2018 13:52	3	0	1	N	236	236	1	4.5	0.5	0.5	0	0	0.3	5.8
5	2	28/11/2018 15:52:25	28/11/2018 15:55	5	0	1	N	193	193	2	3.5	0.5	0.5	0	0	0.3	7.55
6	2	28/11/2018 15:56:57	28/11/2018 15:58	5	0	2	N	193	193	2	52	0	0.5	0	0	0.3	55.55
7	2	28/11/2018 16:25:49	28/11/2018 16:28	5	0	1	N	193	193	2	3.5	0.5	0.5	0	5.76	0.3	13.31
8	2	28/11/2018 16:29:37	28/11/2018 16:33	5	0	2	N	193	193	2	52	0	0.5	0	0	0.3	55.55
9	1	01/01/2019 00:21:28	01/01/2019 00:28	1	1.3	1	N	163	229	1	6.5	0.5	0.5	1.25	0	0.3	9.05
10	1	01/01/2019 00:32:01	01/01/2019 00:45	1	3.7	1	N	229	7	1	13.5	0.5	0.5	3.7	0	0.3	18.5

Figure 6.1.1.5: Screenshot of a Yellow Taxi Trip Data from NYC dataset [13]

[12] https://data.world/cityofchicago/beach-water-quality-automated-sensors
[13] https://data.cityofnewyork.us/Transportation/2019-Yellow-Taxi-Trip-Data/2upf-qytp

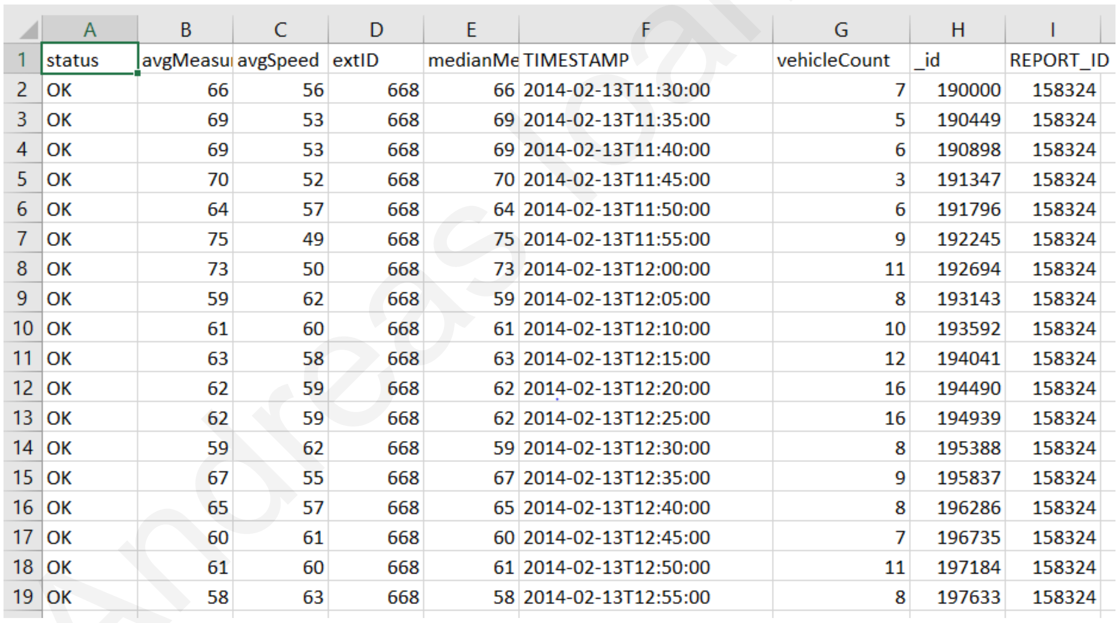


```
"sensorDataConfigs": {
    "sensorPrototypes": [{

            "datasetSensorPrototype": {
                "exportGenerationRate": "true",
                "sensorPrototypeName": "yello trip taxi",
                "messageExportType": "json",
                "timestampFormat": "dd/MM/yyyy HH:mm:ss",
                "timestampedDataset": "true",
                "sortedDataset": "false",
                "datasetFile": "yellow_tripdata_2019-01.csv",
                "timestampColumnName": "tpep_pickup_datetime",
                "sensorIdColumnName": "VendorID"
            }
        }
    ]
}
```

Figure 6.1.1.6: Sensor Modelling Input inside configs_file.json for dataset from

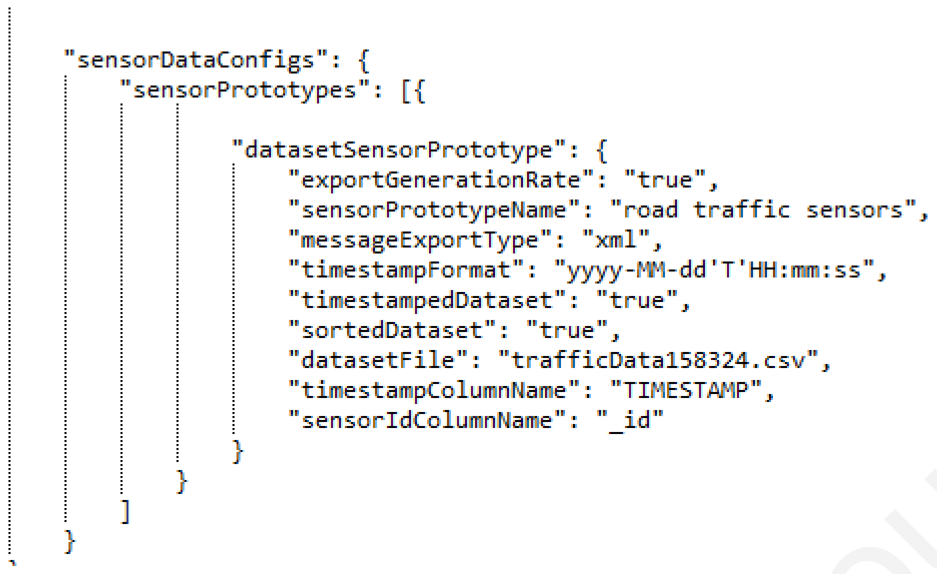
Figure 6.1.1.5



	A	B	C	D	E	F	G	H	I
1	status	avgMeasu	avgSpeed	extID	medianMe	TIMESTAMP	vehicleCount	_id	REPORT_ID
2	OK	66	56	668	66	2014-02-13T11:30:00	7	190000	158324
3	OK	69	53	668	69	2014-02-13T11:35:00	5	190449	158324
4	OK	69	53	668	69	2014-02-13T11:40:00	6	190898	158324
5	OK	70	52	668	70	2014-02-13T11:45:00	3	191347	158324
6	OK	64	57	668	64	2014-02-13T11:50:00	6	191796	158324
7	OK	75	49	668	75	2014-02-13T11:55:00	9	192245	158324
8	OK	73	50	668	73	2014-02-13T12:00:00	11	192694	158324
9	OK	59	62	668	59	2014-02-13T12:05:00	8	193143	158324
10	OK	61	60	668	61	2014-02-13T12:10:00	10	193592	158324
11	OK	63	58	668	63	2014-02-13T12:15:00	12	194041	158324
12	OK	62	59	668	62	2014-02-13T12:20:00	16	194490	158324
13	OK	62	59	668	62	2014-02-13T12:25:00	16	194939	158324
14	OK	59	62	668	59	2014-02-13T12:30:00	8	195388	158324
15	OK	67	55	668	67	2014-02-13T12:35:00	9	195837	158324
16	OK	65	57	668	65	2014-02-13T12:40:00	8	196286	158324
17	OK	60	61	668	60	2014-02-13T12:45:00	7	196735	158324
18	OK	61	60	668	61	2014-02-13T12:50:00	11	197184	158324
19	OK	58	63	668	58	2014-02-13T12:55:00	8	197633	158324

Figure 6.1.1. 7: Road traffic data instance from a dataset for a city In Denmark [14]

[14] http://iot.ee.surrey.ac.uk:8080/datasets.html



```
    "sensorDataConfigs": {
        "sensorPrototypes": [{

                "datasetSensorPrototype": {
                    "exportGenerationRate": "true",
                    "sensorPrototypeName": "road traffic sensors",
                    "messageExportType": "xml",
                    "timestampFormat": "yyyy-MM-dd'T'HH:mm:ss",
                    "timestampedDataset": "true",
                    "sortedDataset": "true",
                    "datasetFile": "trafficData158324.csv",
                    "timestampColumnName": "TIMESTAMP",
                    "sensorIdColumnName": "_id"
                }
            }
        ]
    }
```

Figure 6.1.1.8: Sensor Modelling Input inside configs_file.json for dataset from

Figure 6.1.1. 7

In figures 6.1.1.1, 6.1.1.3, 6.1.1.5 and 6.1.1. 7 we can see 4 instances from 4 different datasets, for completely different applications, and on figures 6.1.1.2, 6.1.1.4, 6.1.1.6, 6.1.1.8 their corresponding input configurations to the system, provided through the *configs_file.json*. In the first dataset (Figure 6.1.1.1), we can see the first rows of the dataset for an application that measured yearly photovoltaic panels data in Cyprus, while in the second one (Figure 6.1.1.3), beach water quality is been measured by some sensors at Chicago Park District beaches. In the third dataset (Figure 6.1.1.5), a screenshot from the Yellow Taxi Trip Data from NYC dataset can be seen and finally in the Figure 6.1.1. 7 exported road traffic data from a city in Denmark. It is obvious that all the datasets are completely irrelevant and different between them and each one having different fields, different timestamp formats, even different types for the values for each field, but still, our system can replay all of them easily, by just providing the correct input configs_file.json accordingly to the dataset. On top of that, even the same dataset could be replayed, with different input configs (maybe different message format or output protocol). This experiment proves that our system can handle multiple and different dataset reproduction using our realization model.

6.1.2 Accuracy Evaluation

Along with the heterogeneity, accuracy is equally important for an IoT workload generator. Not only the accuracy in terms of the data exported to be as real as the real sensor messages, but also accuracy in the time of the data production, that also should be exact as the real ones, in both of the 2 features provided (produce mock data, replay data from dataset).

In order to demonstrate the accuracy, for both time and generated data, let us consider the input at Figure 6.1.2 1. For this scenario, we have a temperature mockSensorPrototype, that has 2 temperature sensors, each one of them generating a new message after 2 seconds. Each generated message consists of a field called temperature that follows a normal distribution, with mean 20 and deviation 5. After 5 seconds from the execution of the scenario, there is a fire that affects the sensor with id 1, making it change the normal distribution it follows, to a new one with mean 40 and deviation 3 for a duration of 5 seconds. *(The values and input configurations of this scenario are just indicative and not real measured for real fire and temperature sensors - Their purpose is to serve the prove of concept of the accuracy of the system in terms of correct messages generation)*

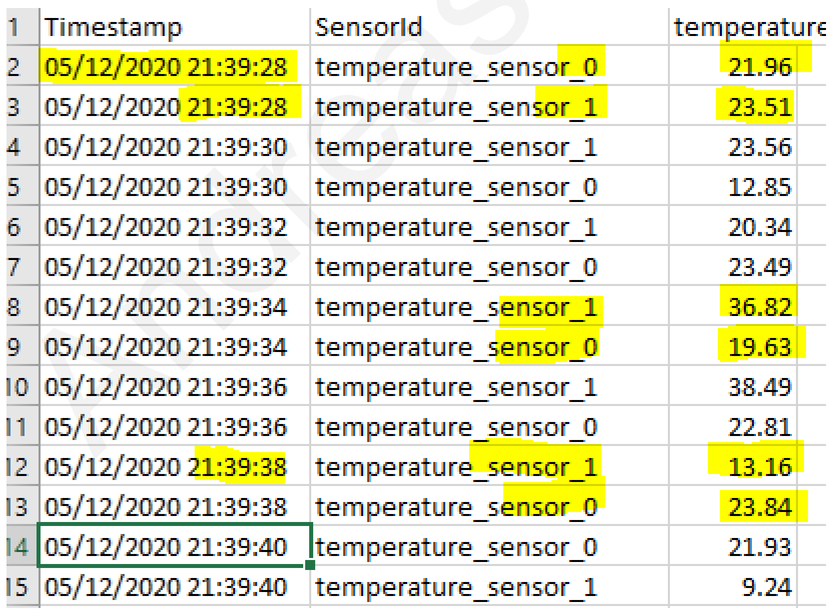
```
{
    "configs": {
        "protocol": "HTTP",
        "protocolConfigs": {
            "requestURI": "/testing",
            "httpServers": [{
                "serverIp": "localhost",
                "serverPort": "8095"
            }]
        },
        "sensorDataConfigs": {
            "sensorPrototypes": [{

                "mockSensorPrototype": {
                    "sensorPrototypeName": "temperature_sensor",
                    "evaluateFieldGenerationRate": true,
                    "outputFile": "temperature_output.csv",
                    "sensorsQuantity": 2,
                    "generationRate": {
                        "constant": {
                            "value": 2
                        }
                    },
                    "messagePrototype": {
                        "type": "xml",
                        "fieldsPrototypes": [{
                            "name": "temperature",
                            "type": "double",

                            "value": {
                                "normalDistribution": {
                                    "mean": 20.0,
                                    "deviation": 5.0
                                }
                            }
                        }]
                    },
                    "scenarios": [{
                        "sensorId": "1",
                        "scenarioName": "fire at room 1",
                        "scenarioDelay": "5",
                        "scenarioDuration": "5",
                        "scenarioFieldValueInfoList": [{
                            "sensorFieldScenarioName": "temperature",
                            "sensorFieldScenarioGenerationRate": {
                                "value": {
                                    "normalDistribution": {
                                        "mean": 40.0,
                                        "deviation": 3.0
                                    }
                                }
                            }
                        }]
                    }]
                }
            }]
        }
    }
}
```

Figure 6.1.2 1: Input file that illustrated the data for the sensor message accuracy validation

In the Figure 6.1.2 2 we can see the first rows that depicts the instance of temperatrue_output.csv file, that was exported for the above scenario. The first messages started to be produced at 21:39:28, and both sensors produced temperatures that followed the normal distribution declared. Then, after 5 seconds (21:39:33) where the fire scenario happens, only the sensor 1 will start producing temperature based on the scenario normal distribution (sensor with id 0 will keep following the initial normal distribution), because the sensor with sensorId 1 has to follow the scenario values generation (that follows different normal distribution with mean 40 and deviation 3). Finally, after 5 more seconds from the fire (10 seconds form the beginning of the execution - 21:39:38, the fire scenario stops), and the temperature sensor 1, returned into producing temperatures based on the initial normal distribution. The values from the exported file prove the accuracy based on the input fields and the scenario provided.



1	Timestamp	SensorId	temperature
2	05/12/2020 21:39:28	temperature_sensor_0	21.96
3	05/12/2020 21:39:28	temperature_sensor_1	23.51
4	05/12/2020 21:39:30	temperature_sensor_1	23.56
5	05/12/2020 21:39:30	temperature_sensor_0	12.85
6	05/12/2020 21:39:32	temperature_sensor_1	20.34
7	05/12/2020 21:39:32	temperature_sensor_0	23.49
8	05/12/2020 21:39:34	temperature_sensor_1	36.82
9	05/12/2020 21:39:34	temperature_sensor_0	19.63
10	05/12/2020 21:39:36	temperature_sensor_1	38.49
11	05/12/2020 21:39:36	temperature_sensor_0	22.81
12	05/12/2020 21:39:38	temperature_sensor_1	13.16
13	05/12/2020 21:39:38	temperature_sensor_0	23.84
14	05/12/2020 21:39:40	temperature_sensor_0	21.93
15	05/12/2020 21:39:40	temperature_sensor_1	9.24

Figure 6.1.2 2: Exported first rows data from the temperature_output.csv file exported by the workload generator for the data accuracy validation experiment based on input in *Figure 6.1.2 1*

6.1.3 Extensibility, Configurability, Customization Evaluation

It is quite obvious, and it can be already observed, that the system promotes and encourages extensibility, and the customization for the inputs. In the sections 5.2.1 and 5.2.2 , the ease of extending both input sensor Prototype features and output protocols can be seen. The provision of the interfaces, and the abstract methods, along with the detailed documentation for the purpose of each method, make any developer able to extend and implement functionalities fulfilling his personal application needs. On top of that, customization in an input dataset can be applied, giving the user the ability to use only the needed properties or data for the dataset, and create extra mock data based on his preferences. Finally, the input properties file (*workloadGenerator.properties)*, and the spring boot framework, where the whole system is built on, can provide configurability to the user, even for different deployments for the same sensor input, to manipulate the execution run based on the machine available resources through the input *workloadGenerator.properties* file that stores key-value pairs of system/deployment/execution properties and the user desired values.

6.1.4 Usability evaluation, through Integration with real Fog Emulator on a real scenario use case

In this section, we integrated the workload generator with a real Fog Computing emulator, named the Fogify [18] which is introduced in section 2.1.4 , to illustrate how useful a workload generator is for a testbed and how easily researchers are capable to evaluate different hypotheses. Again, our use case is inspired by smart homes, but in this scenario, a company manages a set of smart-homes, and it has deployed monitoring sensors to observe them. Every sensor disseminates a set of metrics to a region-based broker queue, and then, an operator can submit streaming queries on top of a processing engine deployed on Fog. We utilized a high-performance queue, namely Kafka [15], as a broker and StreamSight [30] as the processing framework. StreamSight is built on top of Apache Spark [16] and is a streaming processing Framework for edge computing analytics. The Figure 6.1.4 1 presents how the real-world topology for the scenario would be visualized, in which each region would generate its own sensor data, using different input configs file for its workload generator, and the StreamSight will process all the received stream from all regions), while Figure 6.1.4 2 illustrates the emulation, topology happened in our experiment using Fogify deployed on a single virtual machine!

The topology of the experiment is described by following Fogify's model, and Figure 6.1.4 2 illustrates a high-level overview of it. Fogify will handle the instantiations and the interconnection of the other containerized services (Kafka, workload generator and Spark with StreamSight) on deployment using Fogify’s docker-compose specification extension and then it will emulate all the above services connected under the same network with properties and will distribute resources per service, while the user needs to provide only the queries for the StreamSight and the input configs for the workload generation (in this experiment we have just one instance – for one region, but it could be easily scaled to more instances, with the same way).

[15] https://kafka.apache.org/documentation/
[16] https://spark.apache.org/docs/latest/

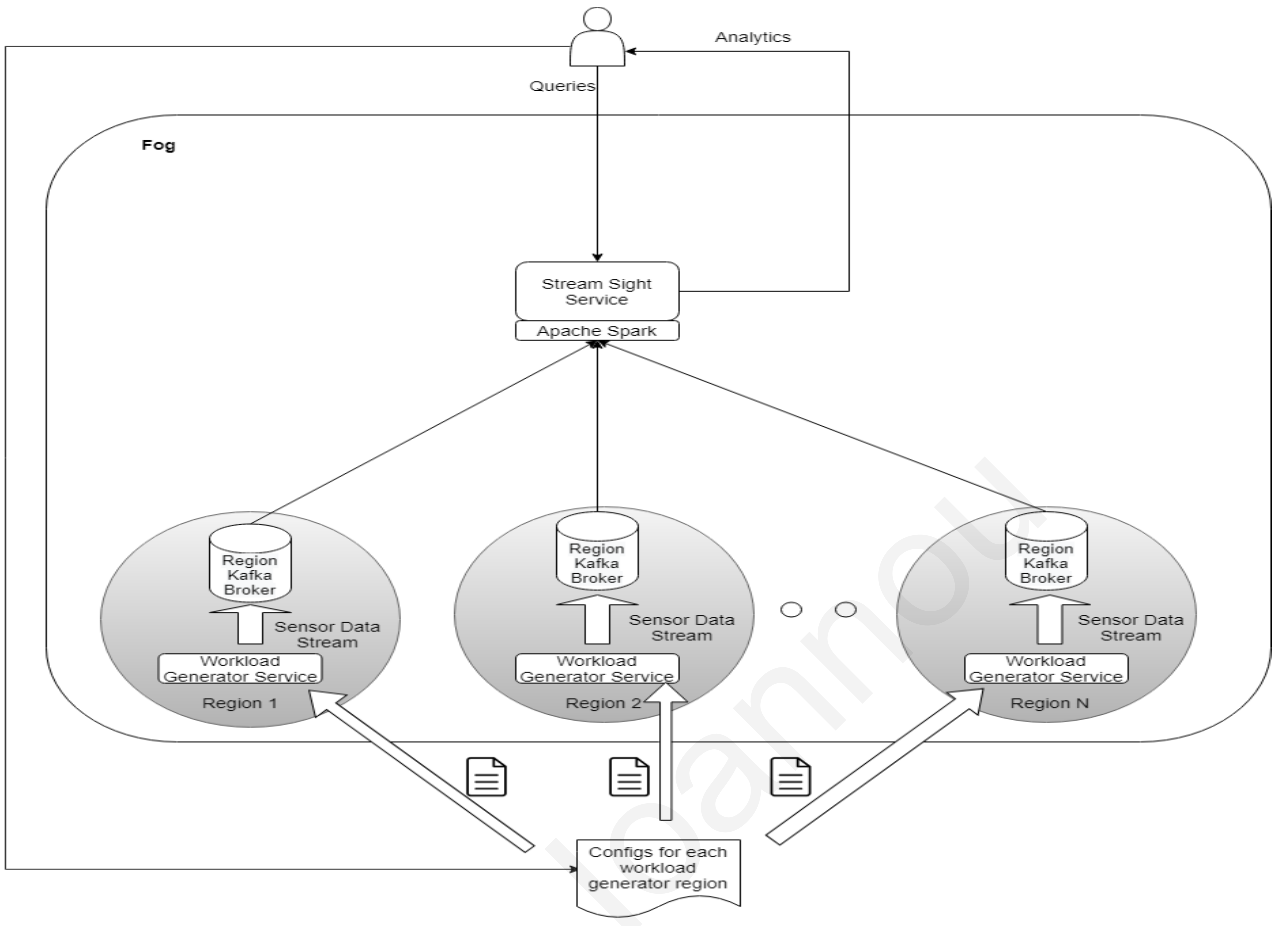


Figure 6.1.4 1: Real topology of scenario

More accurately for the experiment, all services have restricted processing capabilities (2cores at 1.4GHz with 2GB ram) and are connected to the same network with a network delay and bandwidth between the nodes to be 10ms and 100mbps respectively. All services along with the workload generator are containerized and initially described with docker-compose file. All of the experiments are run on top of a Virtual Machine with 16cores and 16GB of RAM and the workload generator is introduced to Fogify as a containerized service.

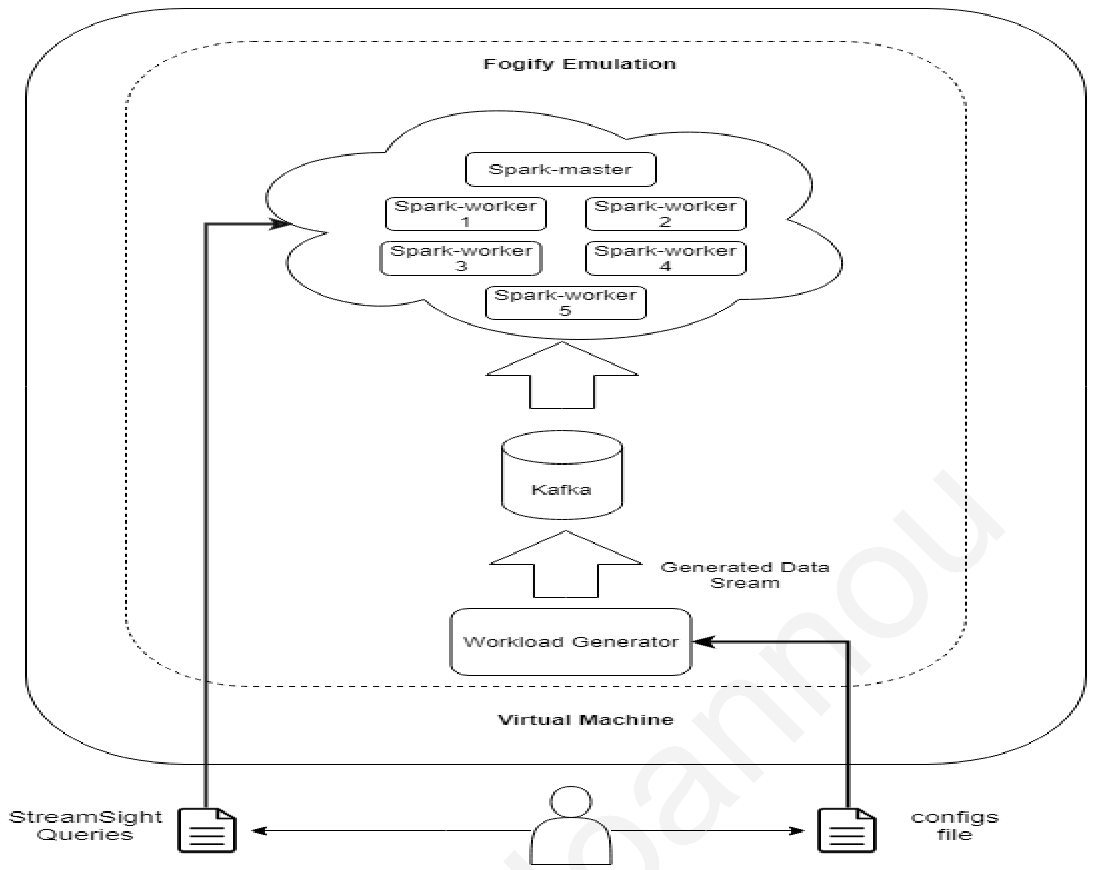
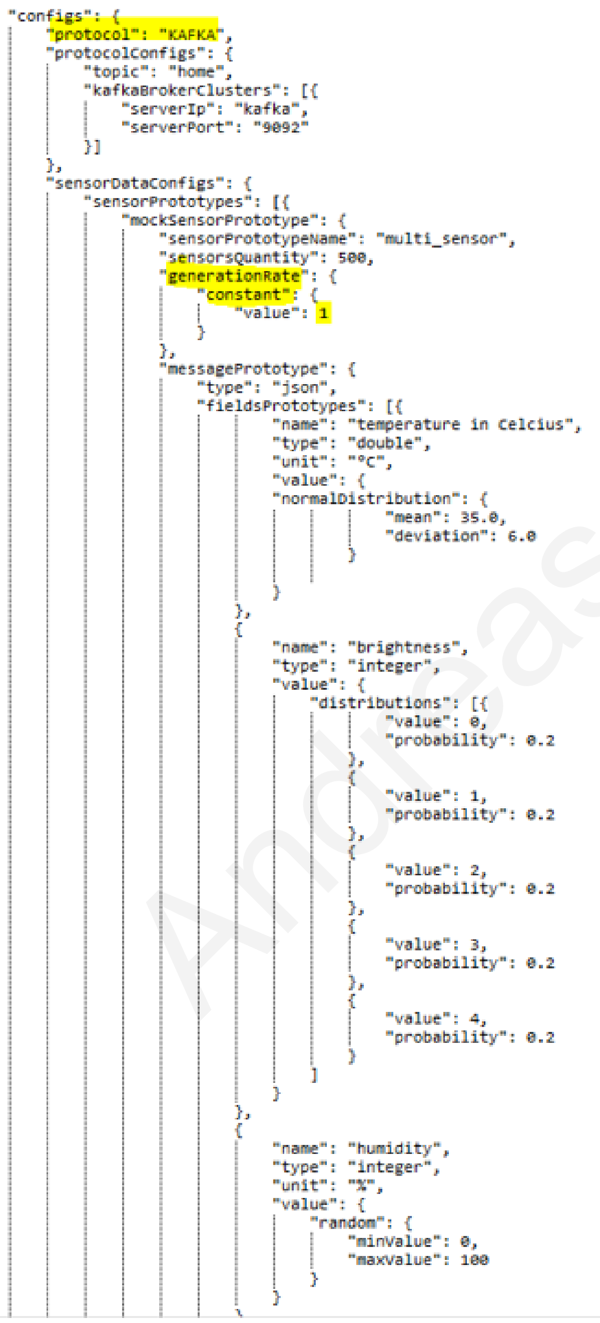


Figure 6.1.4 2: Emulation topology using Fogify on a virtual machine for the experiments

Now that everything is deployed, and the spark using his worker cluster is ready to process the queries, we are ready to run the experiment. To this end, the operators' evaluation is twofold, firstly, they evaluate the scalability of the StreamSight by increasing periodically the number of sensors and measure the processing latency of the streaming engine, and secondly, they evaluate the effect of the data skewness on the performance. The skewness of values is known in distributed processing systems since the partitioning algorithms distribute the keys by following hash functions, thus some workers may force to process much more data than others.

The base workload generator input file used for the 2 experiments can be seen in Figure 6.1.4.1 1. For this set of experiments, we have KAFKA broker, hence the KAFKA protocol and the KAFKA protocolConfigs, and a general sensor prototype, where each sensor disseminates temperature, brightness and humidity. Temperature follows a normal distribution with mean 35 and deviation 6, brightness can have an integer value in range [0..4], where 0 implies dark and by increasing the value we reach 4 which is shining, and all of the values of brightness following a probabilistic distribution (20 % for each one). Finally, we have humidity that is a random generated integer between [0..100]. Each sensor generate data per 1 second.



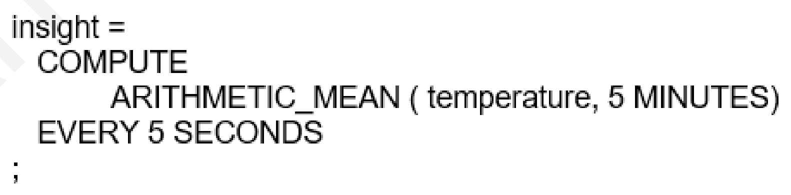
```
"configs": {
    "protocol": "KAFKA",
    "protocolConfigs": {
        "topic": "home",
        "kafkaBrokerClusters": [{
            "serverIp": "kafka",
            "serverPort": "9092"
        }]
    },
    "sensorDataConfigs": {
        "sensorPrototypes": [{
            "mockSensorPrototype": {
                "sensorPrototypeName": "multi_sensor",
                "sensorsQuantity": 500,
                "generationRate": {
                    "constant": {
                        "value": 1
                    }
                },
                "messagePrototype": {
                    "type": "json",
                    "fieldsPrototypes": [{
                            "name": "temperature in Celcius",
                            "type": "double",
                            "unit": "°C",
                            "value": {
                            "normalDistribution": {
                                        "mean": 35.0,
                                        "deviation": 6.0
                                    }
                            }
                        },
                        {
                            "name": "brightness",
                            "type": "integer",
                            "value": {
                                "distributions": [{
                                        "value": 0,
                                        "probability": 0.2
                                    },
                                    {
                                        "value": 1,
                                        "probability": 0.2
                                    },
                                    {
                                        "value": 2,
                                        "probability": 0.2
                                    },
                                    {
                                        "value": 3,
                                        "probability": 0.2
                                    },
                                    {
                                        "value": 4,
                                        "probability": 0.2
                                    }
                                ]
                            }
                        },
                        {
                            "name": "humidity",
                            "type": "integer",
                            "unit": "%",
                            "value": {
                                "random": {
                                    "minValue": 0,
                                    "maxValue": 100
                                }
                            }
```

Figure 6.1.4.1 1: Base Input file of workload generator for scenario

6.1.4.1 Increasing Workload Test

In the first experiment, the operators submit a StreamSight query (Figure 6.1.4.1 2) that computes the overall average temperature of all sensors. Basically, the translation of the query is to calculate the average temperature for all sensors, by processing the sensor messages every 5 seconds and calculating the average temperature, using the sliding window technique on a window of 5 minutes. The system will be collecting data stream from the Kafka (generated from the workload generator) and per 5 seconds it will do the processing based on the collected data from the previous 5 minutes, and finally after 10 minutes of the experiment, it will output the results.

Since the operator needs to evaluate a various number of sensors, the operator simply can use the base input configs_file.json in Figure 6.1.4.1 1, and change gradually the sensorsQuantity to the desired number of sensors and submit the model to the generator. Specifically, the operator instantiates 500, 1000, 1500 sensors in each experiment, each experiment runs for 10 minutes, and every sensor sends a datapoint with temperature, humidity, brightness, etc., every second.



```
insight =
  COMPUTE
         ARITHMETIC_MEAN ( temperature, 5 MINUTES)
  EVERY 5 SECONDS
;
```

Figure 6.1.4.1 2: StreamSight Query for the average temperature of 5 minutes

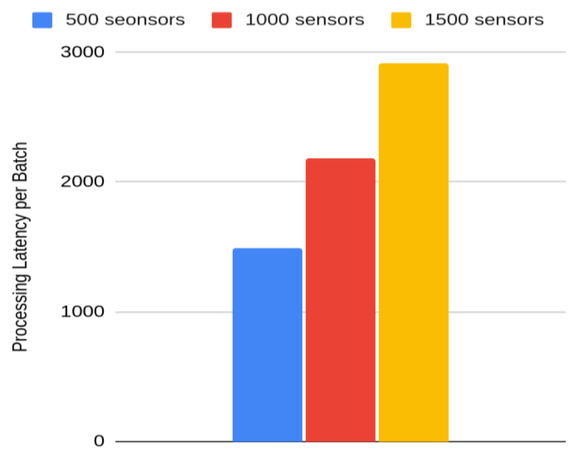


Figure 6.1.4.1 3: Process latency based on the sensor number

Figure 6.1.4.1 3 depicts the Spark's processing latency in milliseconds for each quantity of sensors. It is obvious that the number of sensors, and consequently the amount of the data, dictates the streaming processing latency, since by increasing the sensorsQuantity, we will have more sensors which implies more sensor messages that need to be processed in the same period (5 second). With this experiment, the operator can be sure that the system will provide acceptable performance between 1000-1500 sensors, since even with 1500 sensors, the latency (almost 3 seconds) don't exceeds the 5 seconds that is the interval processing time for the calculation of the query.

In a deeper analysis, Figure 6.1.4.1 4 illustrates the processing delay points for each number of sensors. Each point represents the processing latency for each batch, with the input tuples number equal to the sensor number (each sensor produces 1 sensor message per second). With this, we can observe the distribution of the processing delay for each sensor quantity. Specifically, even if the 500 sensors have the lowest processing delay, they have the largest deviation among the experiments. Given that the processing latency should be less than the execution interval (in order for the system to be stable and do not build backpressure - 5 seconds in this experiment), the operators know that they cannot reduce the interval due to the unexpected deviation even in a low number of sensors.

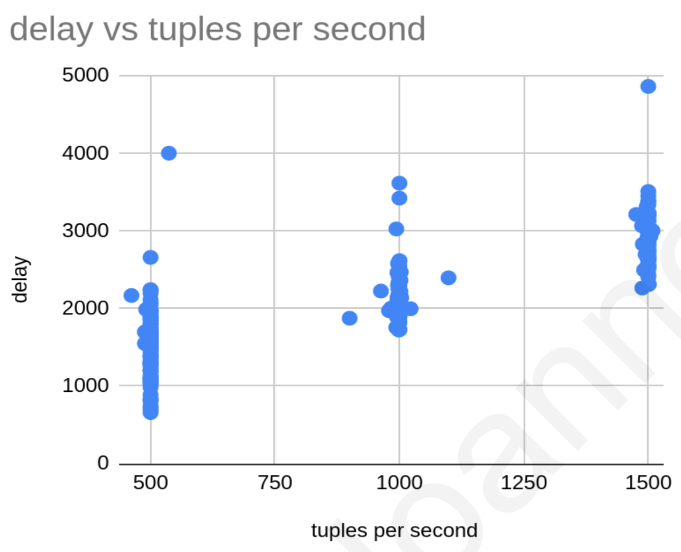


Figure 6.1.4.1 4: Latency of processing for each batch based on the sensors number(sensor messages number)

6.1.4.2 Data Skewness Experiment

After the first experiment, users would like to evaluate how skewed values influence the performance of a processing engine. For that reason, they updated the previous query and group the average of the temperature by the brightness measurements (query in Figure 6.1.4.2 1). In order to provide skewed data, they used the base input (Figure 6.1.4.1 1) configs_file.json, but instead of changing the sensorsQuantity they changed the brightness field probabilities distributions. The brightness as explained before, is measured by ordinal values, from 0 (dark) to 4 (shinning). By following the generator modelling, the operators could describe the possibility of each group in the description of the sensors. In the first experiment, the brightness values are equally distributed in every group (0: 20%, 1: 20%, 2: 20%, 3: 20%, 4: 20%), and in the second experiment, the values of the brightness are skewed (0: 5%, 1: 5%, 2: 5%, 3: 5%, 4:80%).

```
insight =
  COMPUTE
        ARITHMETIC_MEAN ( temperature, 5 MINUTES)
  BY brightness EVERY 5 SECONDS
;
```

Figure 6.1.4.2 1: StreamSight Query for the average temperature of 5 minutes, grouped by brightness

Figure 6.1.4.2 2 depicts the average processing delay between skewed and non-skewed values. We observe around 25% more processing delay when the workload has high skewness. This is reasonable since the data partitioning algorithms of big data engines, like Spark, do not take into account the size of the data and only hash the partition key to decide where the data should be placed.

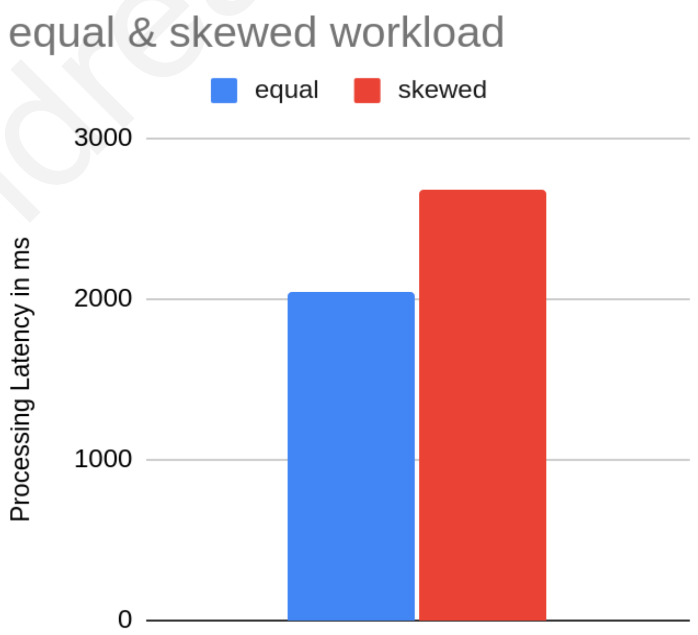


Figure 6.1.4.2 2: Results of experiment for skewed workload

In conclusion for this set of experiments, the alterations of the sensor values and sensor quantity are crucial for the application performance evaluation. Furthermore, application owners can reveal hidden insights from their services and evaluate large-scale scenarios. Finally, the IoT workload generators are essential building blocks for other systems, like fog & edge simulators and emulators, that currently use general-purpose data generators.

6.2 Performance Evaluation and limitations

In this section, after some experiments using different factors, the performance of the system, in combination with the system limitations, that affect the performance will be stated.

Before we move to the experiments, it is important to state and define what are the factors that determine the performance of our system. The performance of our system can be measured in 2 ways, both under the same requirement pillar, the accuracy. It is obvious that one of the performance evaluation factors is the accuracy of the data exported, in order to always follow the provided input sensor model configurations (e.g.: a sensor that has a message field with some random range value, can’t generate value for this field that exceeds the declared input range at any case - except in scenarios mode). This would be a violation in the system accuracy, for the sensor messages accuracy. The other factor that the accuracy can be measured is the frequency that each message is being produced. The user provides the desired sensor message generation rate (the interval time in seconds between each sensor message) or the dataset that will be replayed using some timestamp column, and he expects the data to be generated in the correct time while testing his application. More precisely, the user would demand and want the data to be generated at the correct and real times with ideally zero or extremely small delay, that doesn't affect his application's evaluation. The performance regarding the sensor messages accuracy can be seen in section 6.1.2 .

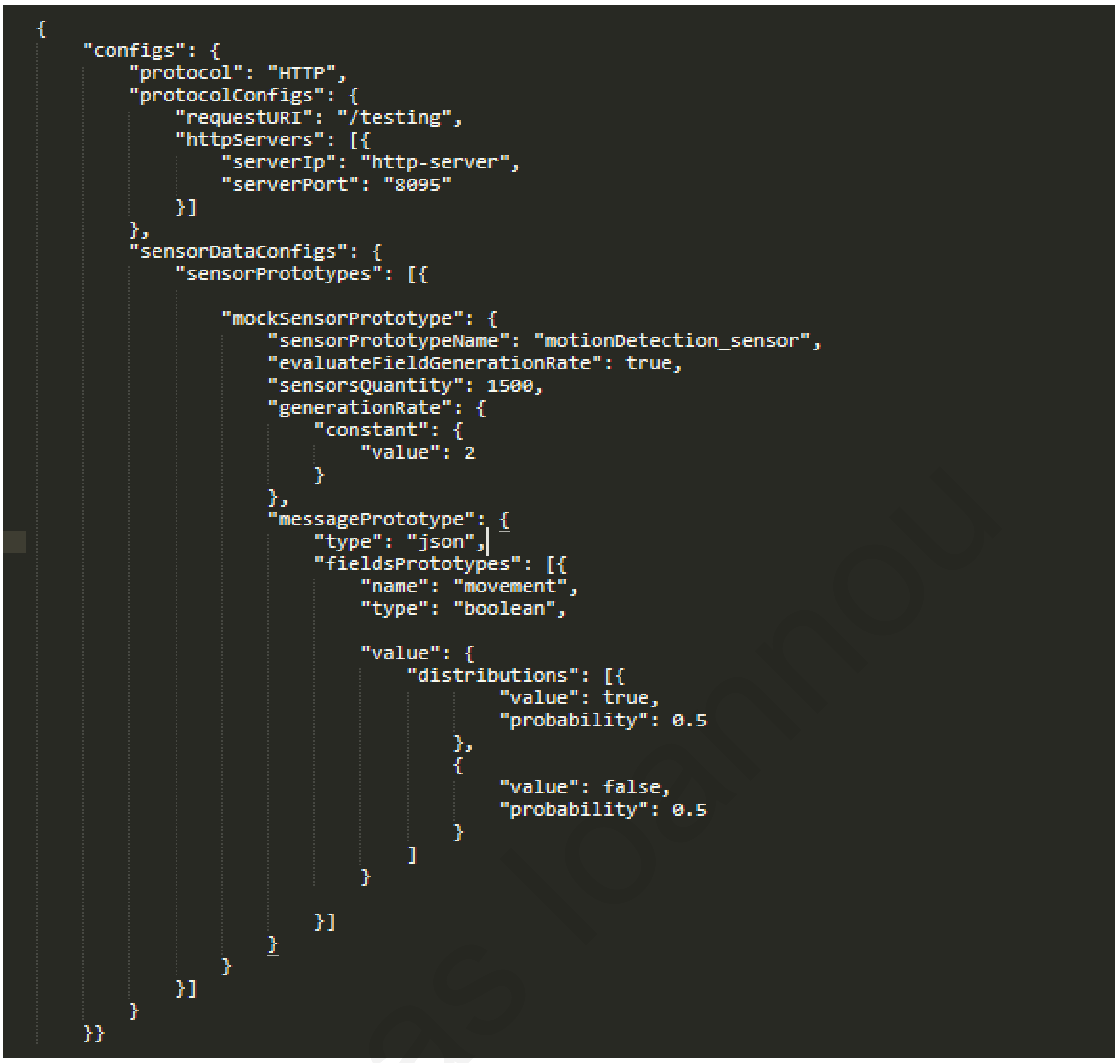
6.2.1 Sensor Message Delay

In this section the performance based on the sensor messages delay will be evaluated. By sensor message delay, we refer to the case that the frequency between 2 successive messages is greater than the expected interval time. For instance, if the generation rate for one mockSensorPrototype is constant value of 2 seconds, and the first message was sent at 10:00:00 and the next sensor message (by the same sensor) was sent at 10:00:04, this indicates that the delay was 2 seconds, because the system was supposed to generate the second message at 10:00:02. Of course, the delay can be accepted in some cases, that tends to zero, or generally doesn't affect the testing of the user, so we have to set a rule in order to be able to test this factor. The most reasonable rule, in order for an average delay to be considered performance downgrade, it is not to exceed the required sensor message interval provided by the user, since this will stack the requests on the user application. For example, if the user provided that he wants the generation rate frequency to be 3 seconds, the threshold of supported input sensors, would be until sensors start to produce sensor messages with average delay exceeding the desired frequency (3 seconds in this example).

6.2.1.1 Input and Evaluation Methodology

Before moving to the experiments and the results let us first introduce the methodology used in order to evaluate the system performance, based on the sensor message delay. For this testing, in order to have an accurate evaluation and to isolate any external factors that could affect the evaluation and the machine performance, we make the input to be as simple as it can be, in order to cover the evaluation scenario and not to create overhead to the machine for anything else beyond the factor being tested.

In the Figure 6.2.1.1 the testing scenario input can be seen. As it is previously noted, we want to make the executions as simple without any external factor to affect the evaluation, hence we have just one mockSensorPrototype and one field, and no export to output file or scenarios since this would create extra overhead due to more threads running. The only things that will be changing (according to the factors and the experiments) are the *sensorsQuantity* value and the *generationRate* value.



```
{
    "configs": {
        "protocol": "HTTP",
        "protocolConfigs": {
            "requestURI": "/testing",
            "httpServers": [{
                "serverIp": "http-server",
                "serverPort": "8095"
            }]
        },
        "sensorDataConfigs": {
            "sensorPrototypes": [{

                "mockSensorPrototype": {
                    "sensorPrototypeName": "motionDetection_sensor",
                    "evaluateFieldGenerationRate": true,
                    "sensorsQuantity": 1500,
                    "generationRate": {
                        "constant": {
                            "value": 2
                        }
                    },
                    "messagePrototype": {
                        "type": "json",
                        "fieldsPrototypes": [{
                            "name": "movement",
                            "type": "boolean",

                            "value": {
                                "distributions": [{
                                        "value": true,
                                        "probability": 0.5
                                    },
                                    {
                                        "value": false,
                                        "probability": 0.5
                                    }
                                ]
                            }

                        }]
                    }
                }
            }]
        }
    }}
```

Figure 6.2.1.1 1: Base Input file for sensor message delay accuracy experiments

Now that we introduced the input and the criteria, it is time to explain how the evaluation will happen, and more precisely how the delay is being measured. In order to be able to calculate the average delay of one mockSensorPrototype, i created an application that receives HTTP requests through the **"/testing"** endpoint from the workload generator (mock sensor messages), and process them accordingly, in order to calculate the average delay for one mockSensorPrototype. We prepare the HTTP application in order to know which sensor Prototype name should process. More precisely, the preparation for the HTTP testing application happens with an API call. Through that API call, we inform the application with 2 things, 1) which sensor prototype name should only be processed (let us name this input as ***evaluationSensorPrototypeName***) and 2) which is the expected generation rate for this sensor prototype (the interval message generation frequency for the sensors of this sensor prototype - let us name this input as ***expectedGenerationRate***). Then, the application internally, when it is being prepared, creates a thread that receives these 2 inputs, and has a blocking queue, where the input messages will be stored with priority the timestamp. In the continuation, when the HTTP application receives a POST request from the workload generator (the request is the sensor message packaged along with the sensorId and the timestamp the request was sent form the workload generator, in the request payload) it puts it to the queue of the evaluation thread, and if the sensorId of the request (the sensorId contains the sensorPrototype name) matches the preparation input sensor prototype name (***evaluationSensorPrototypeName)***, then the message will be processed else it will be ignored. After the evaluation thread filters the messages from its queue, it stores the necessary information for each message in order to use them to calculate the delay. The idea here is that, from the received packaged message, the thread keeps only the ***timestamp*** that the request was sent, the ***received messages count*** and the ***sum of the average delay*** for ***each sensor id***. That way, the evaluation thread, knows at any time what is the *current latest message timestamp for each sensorId* , and when it receives a new message from the same sensor (same sensorId) it calculates the difference of the timestamps between the previous and the last messages of this sensor and checks if the difference is greater than the expected generation rate that the application had as input in the preparation (***expectedGenerationRate)***. If the difference between the previous latest message of the same sensor and the current last message received is greater than the expected-declared interval generation rate time, then it sums this delay, and increases the requested messages counter for this sensor by 1. After the evaluation duration finishes, the thread calculates the average delay for each sensor (the sum of the delays divided by the requests count for the sensor) and when it has calculated the average delay for each sensor, it calculates also the average delay for all of the sensors (the sum of the average delays for all sensors for the specified sensor prototype divided by the sensors quantity) . The output of this equation will be considered the average delay for the testing scenario instance.

$$\frac{\sum_{i=0}^{x}\left(\frac{\sum_{r_i=0}^{n_i}\left(\left\|t_{last}-t_{prev}\right\|-T_{expected}\right)}{n_i}\right)}{x}$$

,where

t_{last} = timestamp of the last received message for sensor with id = i

t_{prev} = timestamp of the previous received message for sensor with id = i,

$T_{expected}$ = the expected generation rate,

n_i = the number of the received messages for sensor with id = I,

x = the number of the sensors of the sensor prototype

Equation 6.2.1.1 1 : Equation used in order to calculate the average sensor message delay of a MockSensorPrototype

The Equation 6.2.1.1 1 express how the value of the average delay for one sensor prototype is calculated in the HTTP testing application for each instance of the evaluation.

6.2.1.2 Evaluation Experiments

Now that we have stated and explained the methodology of the experiments for the evaluation of the performance based on sensor message delay, let us see and introduce the experiments and analyse the results.

The different experiments happened based on different testing factors. The 3 factors that were tested are the *input sensor quantity (sensorsQuantity)* for the mock sensor message, the *generation rate (generationRate)* for the mock sensor message, and the *processing power/resources of the hosted machine*. Each experiment will run for 5 minutes, and in the end the average sensor message delay will be calculated for each experiment.

6.2.1.2.1 Sensor Quantity Factor

Probably the number 1 factor that could affect the system performance is the number of mock sensors to be created from the input. Since our system creates a thread for each emulated sensor, it is quite obvious and inevitable that there will be some constraints in the number of sensors that the system can handle, given the fact that no machine has unlimited processing resources in order to be able to process in parallel a huge amount of threads. With a huge input sensor number, which implies huge thread quantity to be created, the system performance could be affected.

For this experiment, we have the input file that can be seen in the Figure 6.2.1.1 1 and the only variable would be the *sensorsQuantity*. We will run the experiment, for different sensors Quantities (500, 1000, 15000, 2000, 25000, 3000) and for each different *sensorsQuantity*, the duration until the end of the scenario and the calculation for the final average sensor message delay using the methodology explained in section 6.2.1.1 will be 5 minutes, which is a period that gives us representative results.

Sensor Number N	**Average Message Delay (s)**
500	0.00732
1000	0.023679
1500	0.026599
2000	0.112249
2500	0.195059
3000	0.518606

Table 6.2.1.2.1 1: Experiment results for input sensors number factor

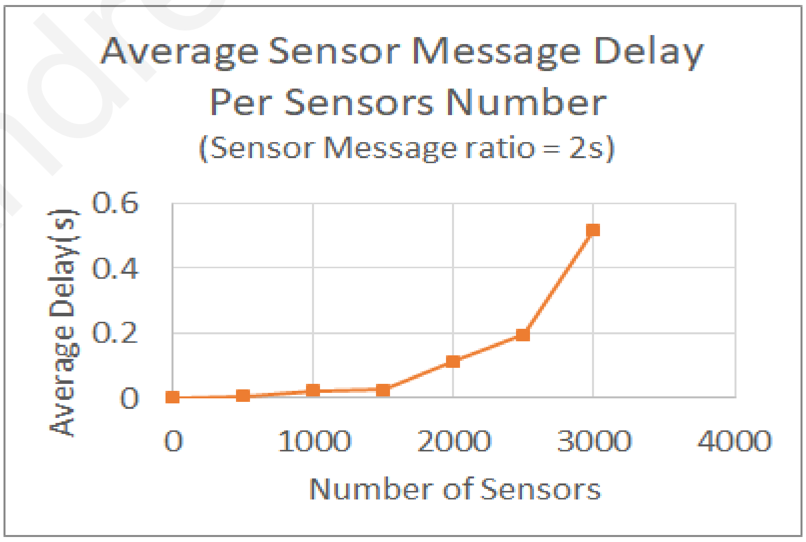


Figure 6.2.1.2.1 1: Graphical representation based on the results in Table 6.2.1.2.1 1: Experiment results for input sensors number factor for the sensor number factor experiment

In Table 6.2.1.2.1 1 we can see the results for the evaluation of the system performance by factoring the number of sensors. In the first column is the number of sensors and in the seconds column, the average sensor message delay for the corresponding input number of sensors in a machine that hosted the application, within a docker container of 2 cores and 2 GB of RAM. Also, the Figure 6.2.1.2.1 1 depicts the corresponding graphical representation for the measured results of the table. The first thing we notice is that the average delay, even for a small number of threads, exists but it is indeed insignificant (~0.007s). Then, it can be observed clearly that as the number of sensors is increasing, the sensor message delay is also increasing smoothly, until it reaches the 2000, then it grows faster. Finally, for 3000 sensors, the average delay is still way under the generation rate (2 seconds limit we set above).

In addition, the figures 6.2.1.2.1 2, 6.2.1.2.1 3, 6.2.1.2.1 4 illustrate the CPU usage using the cadvisor[17] container in the testing hosted machine, that can create statistics and analyse the CPU at the container that the application runs. In the figures, we can see an instance of the CPU usage (percentage) per time for the docker container that the application was hosted while running the experiment for a different number of sensors. In the first figure(6.2.1.2.1 2) that represents an instance of CPU usage for sensorsQuantity = 1000, it can be seen that the usage is at low levels, by using on average only 22% of the available CPU cores. Then, In the Figure 6.2.1.2.1 3 where the input sensors number = 2000 we can see that the CPU usage is increased, and fluctuates between 30 - 45%, and finally at the Figure 6.2.1.2.1 4 the CPU usage can reach the 70%, when the input sensors number = 3000.

[17] https://github.com/google/cadvisor

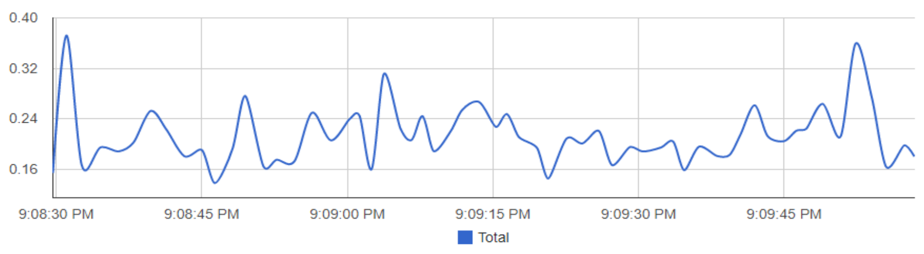


Figure 6.2.1.2.1 2: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 1000

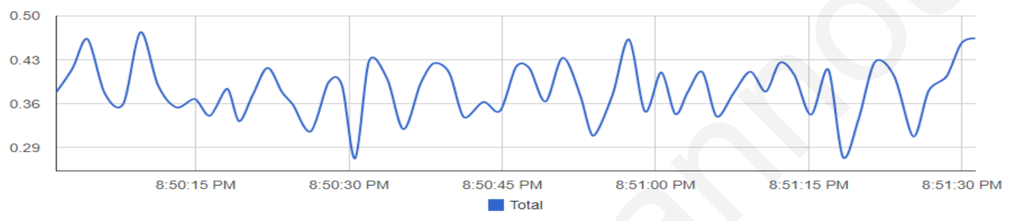


Figure 6.2.1.2.1 3: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 2000

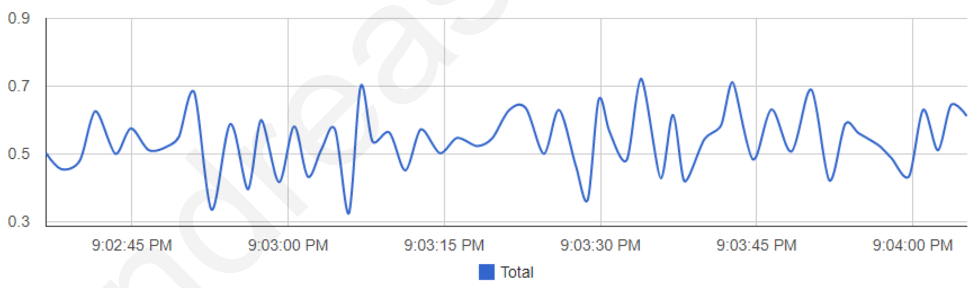


Figure 6.2.1.2.1 4: CPU Usage using cadvisor on the hosted application docker container with 2 cores for input sensors number = 3000

The results which are obvious, and quite expected, show firstly that the bigger the input number of sensors is, the more the performance will be affected. Also even the average delay for a small number of sensors is very small, tending to zero, it exists, and the reasons behind that is the delays for network calls to reach the HTTP application, and the fact that in order to have 0 delay, a machine should be able to handle at the same time all the threads. Of course, this argument is not feasible yet, since the machines can't reach this point yet and they work concurrently (using their multicore architecture) and not parallel, which implies that there will be time that the OS will have to do multiple thread context switches.

Context switch is the procedure when the thread inside the CPU is switched, voluntary (when the thread for example does i/o actions, like waiting for the response for the request or is sleeping state) or nonvoluntary (the CPU forces the thread to be switched because the OS interrupt time has running out, which means the thread is too long in the CPU and it needs to be switched in order for another thread to use it). The reason behind the context switches of threads is that all active threads can't access the CPU at the same time. Regardless of the type of switch, the thread that will be switched out, cannot use the CPU, which implies that it will be waiting idle, until the OS switches it on again to resume its tasks. All this waiting, and the small amount time created due to the overhead of the OS to make the context switches while switching the threads, will be summed and it will produce this delay, which is inevitable.

Back to the experiment, the results verified our theory, and it is easy to see that when the sensor input is bigger, which implies that the threads created by the application and the OS are more, hence we will have more delay, affecting the performance of the workload generator, with one reasonable explanation the increase of context switches.

6.2.1.2.2 Generation Rate Factor

We have seen that the number of sensors affects the performance of the workload generator, which was quite expected, but now let us check another more complex factor. In this experiment the factor that will be tested, is the generation rate for the sensors. For this experiment we will keep the input from

Figure 6.2.1.1 1 and we will test by gradually increasing the generation rate for the sensor messages to 6 and 10 seconds, if it affects the system performance and more accurately the average sensor message delay.

Input Sensors Number (N)	Average Message Delay (s)		
	Expected Generation Rate = 2s	Expected Generation Rate = 6s	Expected Generation Rate = 10s
500	0.00732	0.01241	0.018775
1000	0.023679	0.040051	0.018022
1500	0.026599	0.038017	0.016636
2000	0.112249	0.065697	0.052392
2500	0.195059	0.071678	0.056858
3000	0.518606	0.106777	0.120352

Table 6.2.1.2.2 1: Experiment results for generation rate factor

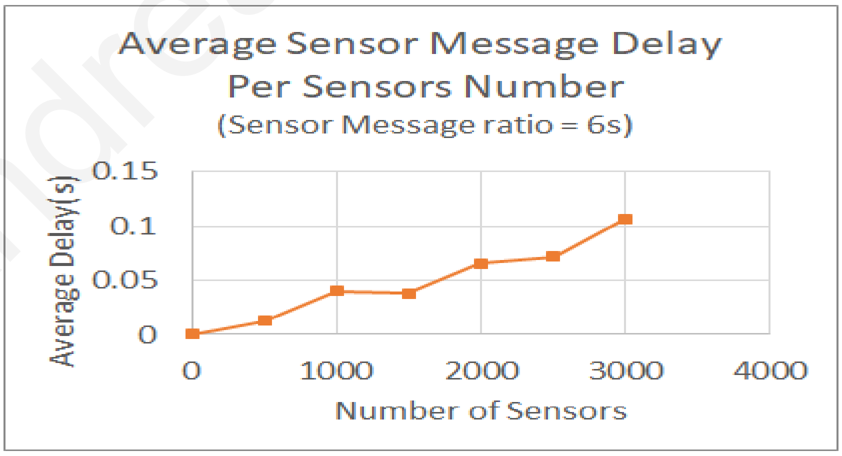


Figure 6.2.1.2.2 1: Graphical representation based on the results in Table 6.2.1.2.2 1: Experiment results for generation rate factor

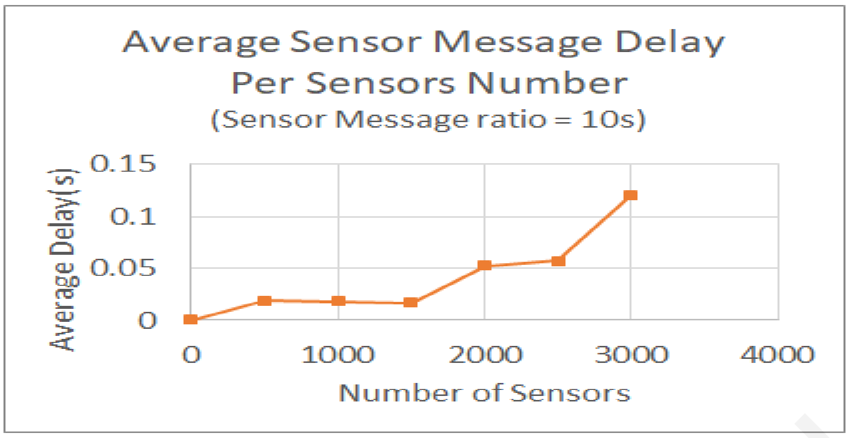


Figure 6.2.1.2.2 2: Graphical representation based on the results in Table 6.2.1.2.2 1*: Experiment results for generation rate factor*

In the Table 6.2.1.2.2 1 we can see the results of this experiment along with the results for the previous. In the first column of the table we can see the results for the previous experiment (with generation rate 2 s) and in the next columns we can see the results for generations rate 6 and 10 s respectively. The figures 6.2.1.2.2 1, Figure 6.2.1.2.2 2 present the corresponding graphical representations for the new statistics.

The results show some patterns which are expected, but there is also an unpredictable behaviour, regarding the factor of generation rate. To begin with, also for different generation rates, the behaviour regarding the increase of sensor number is the same, since we have more average delay, with a bigger number of sensors. Also, between generation rate 2s and generation rate 6s, the performance is way better (~0.5s for 3000 sensors and generation rate 2s and ~0.1s for generations rate 6s for same amount of sensors) when we have generation rate 6s, and one possible explanation would be that the context switches number would be less. The argument that the context switches will be less, comes from the idea that since the interval time between sensor messages is less, the number of context switches will be bigger (more sensor messages will be sent since the interval time is less, implies more i/o).

On the other hand, between generation rate 6s and generation rate 10s, it is not clear which has better performance, since they are in the same level more or less.

6.2.1.2.3 Host Machine Resources Factor

The last factor that we will examine, is the resources of the hosted machine. In the previous 2 experiments the machine used was a pc, with the workload generator as containerized service, with 2 available cores and 2 GB RAM, while in this experiment we will test the same scenarios as before, in a machine with more cores and available RAM, a server with 16 available cores and 16 GM RAM. Our guess is that due to the bigger number of available cores, the results will be better than the other machine with fewer resources.

Input Sensor Number (N)	Average Message Delay (s)		
	Expected Generation Rate = 2s	Expected Generation Rate = 6s	Expected Generation Rate = 10s
500	0.005412	0.016465	0.010333
1000	0.00748	0.016077	0.018917
1500	0.010654	0.018813	0.015203
2000	0.024444	0.019979	0.023761
2500	0.026832	0.018879	0.019488
3000	0.139107	0.021722	0.026162

Table 6.2.1.2.3 1: Experiment results for hosted machine cores factor

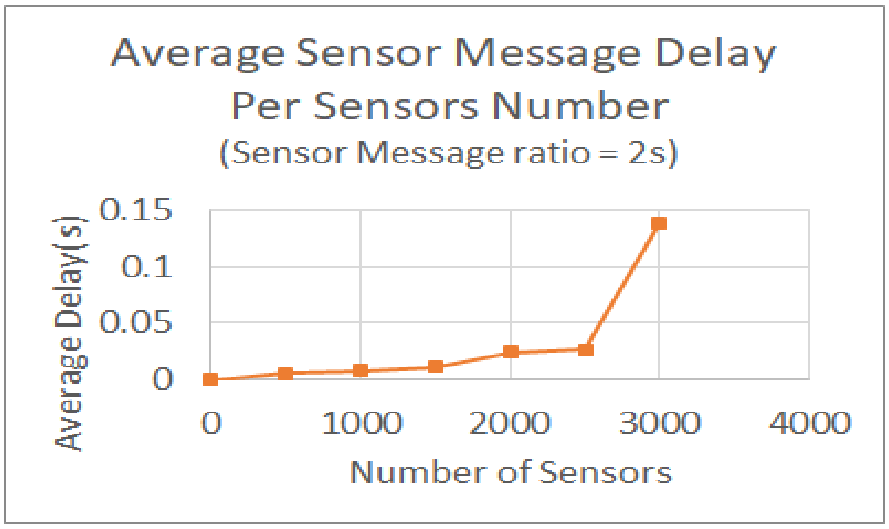


Figure 6.2.1.2.3 1: Graphical representation based on the results in

Table 6.2.1.2.3 1: Experiment results for hosted machine cores factor for generation rate = 2 s in machine with 16 cores

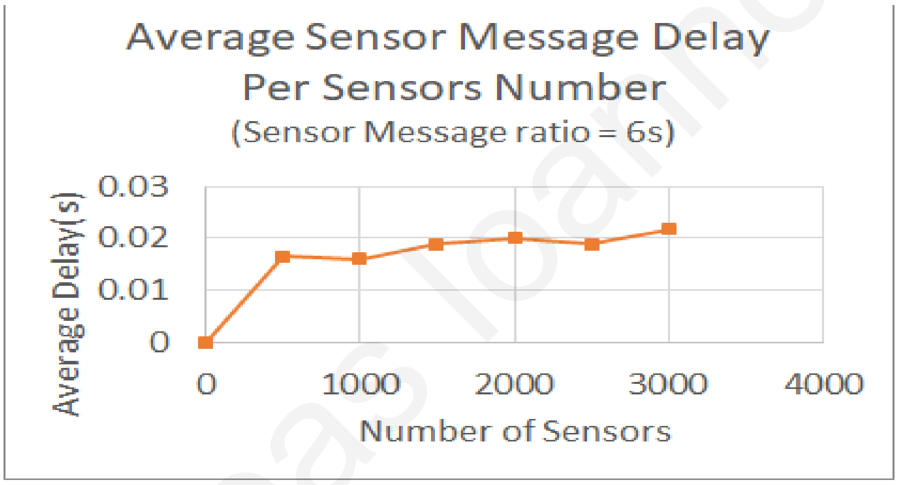


Figure 6.2.1.2.3 2 : Graphical representation based on the results in

Table 6.2.1.2.3 1: Experiment results for hosted machine cores factor for generation rate = 6 s in machine with 16 cores

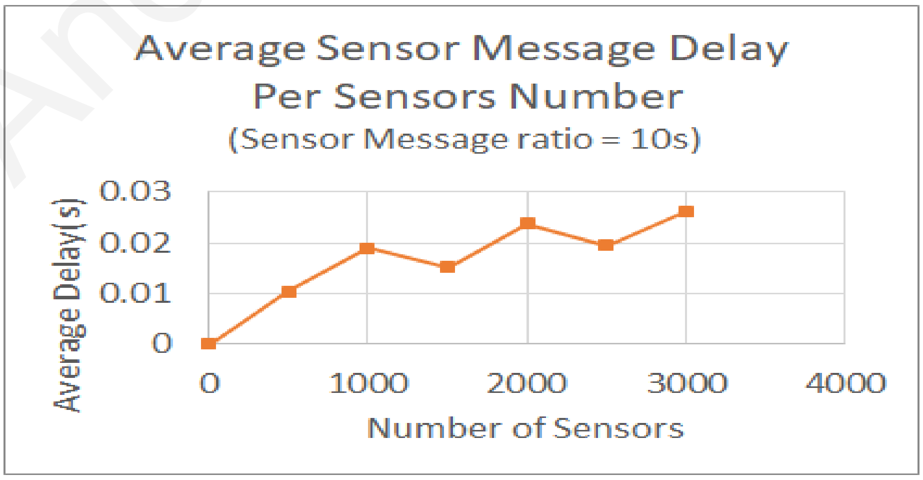


Figure 6.2.1.2.3 3: Graphical representation based on the results in Table *6.2.1.2.3 1: Experiment results for hosted machine cores factor for generation rate = 10 s in machine with 16 cores*

The Table 6.2.1.2.3 1 depicts the results for the same experiments Sensor Quantity Factor, Generation Rate Factor on a more powerful machine, and the figures 6.2.1.2.3 1, 6.2.1.2.3 2 , 6.2.1.2.3 3 its corresponding graphical representations based on the expected generation rate.

The performance overall is better than the machine with fewer cores, since the average delay is less in all the cases, and it is quite expected as the threads can now use more CPU cores, in comparison to the previous tested machine. The more cores a machine can have, it allows more parallel computing which implies more threads working at the same time (16 MAX in this scenario in comparison to the 2 in the previous). Having said that, with more threads using CPU at the same time, the waiting time will be less, hence the average delay will be also less.

Moving to the sensorsQuantity factor, it is needless to say that in this experiment also the sensors number is affecting the performance, since with 500 sensors the average delay for generation rate 2s is ~0.005s, while for 3000 sensors the average delay increases to ~0.13s (which is smaller than the 0.51s on the other machine). The last comment, regarding the generation rate, the behaviour is the same as the previous machine. For generation rate 6s the performance is better than generation rate 2s, and the difference between generation rate 6s and 10 s is not very obvious.

Overall, the results are better, but follow the same patterns as the previous machine.

6.2.1.2.4 Sensors Quantity Limitation

In this experiment we will try to stress the workload generator in order to find its limits. The first thing we would like to find is the limit of how many sensors can the system handle in relation to the cores of the hosted machine, until the system becomes unresponsive or does not function properly. Then, the other goal is to find a threshold of sensor quantity, based on the generation rate, that if the machine passes this threshold, then the delay would be not accepted. In order to find the threshold, we set the limit of accepted sensor message delay to be the same as the generation rate, since, if the system has more delay than this, it will build backpressure, because the requests will queue to the user application and the application won't be able to process them in time.

The attempts will happen on 3 machines, one with 2 available cores, the other with 8 available cores and finally a server with 16 cores. For each of the machines, we will stress the machine in order firstly to find its breaking point and then to find the threshold for the not accepted generation rate (using generation rates 2, 6, 10 seconds).

	Generation Rate 2s	Generation Rate 6s	Generation Rate 10s
Unresponsive Machine sensor limit	8000	8000	8000
Delay exceeds the generation rate sensor limit	4000	-	-

Table 6.2.1.2.4 1: Input Sensors number Performance Limitation on a machine with 2 available cores

	Generation Rate 2s	Generation Rate 6s	Generation Rate 10s
Unresponsive Machine sensor limit	10000	12000	12000
Delay exceeds the generation rate sensor limit	6500	-	-

Table 6.2.1.2.4 2: Input Sensors number Performance Limitation on a machine with 8 available cores

	Generation Rate 2s	Generation Rate 6s	Generation Rate 10s
Unresponsive Machine sensor limit	30000	30000	30000
Delay exceeds the generation rate sensor limit	6000	15000	-

Table 6.2.1.2.4 3: Input Sensors number Performance Limitation on a machine with 16 available cores

On the tables 6.2.1.2.4 1, 6.2.1.2.4 2, 6.2.1.2.4 3 we see the thresholds for the experiments in the machines with available 2, 8 and 16 cores respectively. What we can see is that the only case that the delay exceeds the generation rate is for generation rate 2s, (with exception the machine with 16 cores that we were able to find the limit also for generations rate = 6s) since for bigger generation rates, the machine becomes unresponsive or the containers drop during our task to find the threshold of sensors that the delay exceeds the generation rate. More specifically, it is safe to assume that the maximum number of sensors that the system can emulate in order that the system performance don't affect the users application evaluation is ~6000 sensors for a decent machine (having more than 2 cores), while for machines with 2 cores the threshold is 4000 sensors. Also, for powerful machines having more than 8 cores, the threshold for generation rate bigger than 2s should be around

15000 sensors. The other thing that we wanted to find was the max number of sensors that the system can emulate without having functionality issues or errors based on the available cores of the hosted machine. If we have a look at the table, we see that the limits are more or less independent from the generation rate, and the sensor limit that the system can support starts from 8000 sensors in a weak machine with 2 cores, and grows until the huge number of 30000 sensors in a machine with 16 cores. For a normal machine of 8 cores the limit of sensors that the system can satisfy without any issue is around 10000 sensors.

6.3 Evaluation Results Summary and System limitations

After the experiments and the evaluation happened in the 2 previous sections, it is time to summarize the results, and give an overview of the usability and performance evaluation and the limitations of the system.

Firstly, the usability was evaluated based on the requirements we previously set that a workload generator should have, back to section 3.1. We did prove, that my workload generator could replay completely different applications' exported sensor data from CSV datasets, that encourages the heterogeneity in sensor properties, sensor messages formats and even sensor output protocols. Then, we executed a use case with some sensors and some scenarios that could alter the data population during the data generation, and we confirmed the accuracy of the generated data values based on the input specifications. In addition, a whole sub chapter was dedicated to prove the extendibility of our system, demonstrating how easily the input and output interfaces can be implemented by any developer(section 5.2). Furthermore, we went a step deeper, in order to show our system usability, by integrating it with a real fog emulator, and we illustrated how useful a workload generator is for testbeds or why it could be an essential brick for other systems like edge and fog simulators or emulators and finally how researchers or operators can also use it, not only to evaluate different hypothesis, but also to reveal hidden insight or do data analysis on it.

In addition to the usability, we were determined to find factors that could affect the system performance, which could be translated to the delay between sensor messages for the same sensor. Regarding the accuracy performance due to sensor delay, we first showed that the Sensor Quantity Factor could obviously affect the performance, since the increased number of sensors implies more threads in the host OS, which obviously don’t have unlimited resources to handle huge amounts of threads at the same time, mapping to the needed sensors. Beyond the influence of the input sensor number, the system worked quite decently in a weak machine with 2 available cores, having just ~0.007s average sensor message delay, for input 500 sensors, and generation rate 2s, and the delay is still small until the 3000 sensors, where it reaches the ~0.5s. After that experiment, we figured out that also the generation rate could alter the performance. Basically, the Generation Rate Factor between the sensor messages showed that with greater generation rate, the average sensor message delay potentially could be less, with the explanation that less context switches between the created threads inside the CPU will occur (bigger message interval period implies less generated sensor messages). The last factor that was tested it was the Host Machine Resources Factor. After replaying the same experiments (input sensor quantity along with different generation rates) in a more powerful server with 16 GB RAM and 16 cores, we saw the tremendous difference in the average sensor message delay. The average sensor message delay reached just the ~0.1 s for 500 input sensor messages and generation rate = 2s, being 5 times smaller than the delay found based on same metrics in a machine with 2 cores and 2

GB RAM. The delay was overall smaller than the weaker machine, but the patterns regarding the factors were quite the same. The more sensors the more the accuracy is dropping, and the bigger the generation rate, the smaller is the delay, but rather than that with a decent machine the performance downgrade, can’t seriously affect the users’ application evaluation. Having said that, finally we stressed the system to find its breaking point in terms of performance and generally its limits. After some vigorous stressing to 3 machines with different processing power, we found the limits when the machine’s performance doesn't affect the application regarding the delay. For generation rate 2s we found that the system can provide acceptable delay as long as the sensor input number is less than 4000 for a machine with few resources (2 cores) and around 6000 for decent machines providing CPU with more than 8 cores. In addition, we found out that the system can safely emulate sensors less than 8000, with this limit to reach even the 30000 as the machine power increases, before making the application unresponsive.

Chapter 7

Conclusion and future work

7.1 Synopsis

7.2 Conclusion

7.3 Future work

7.1 Synopsis

In an era that IoT applications are growing, and the deployment using fog computing is emerging, developers are facing various challenges in order to test and evaluate their IoT applications, hence an extra hand of help in the testing phase would be essential and appreciated. Having said that, IoT workload generators should not only provide the required data based on the users provided sensor models and properties, but also support heterogeneity in order to be used with plenty of developers in different scenarios, and make the process of integration with the fog easier. This implementation in comparison to the existing ones, encourages the heterogeneity found in IoT devices, in terms of different output protocols, different sensor message formats and data representations. On top of that, it can be easily extended, and finally using the provided sensor realization model, can comprehend different types of sensors, covering the gap of solutions that are tailored only to specific sensor types or output protocols, lacking generality or cannot be easily extended. The implemented system, provides a real sensors comprehension model, along with an extendable, heterogenous IoT workload generator that can be also integrated with fog emulator and aims to help operators, academic researchers, users or developers to test, deploy, analyse and evaluate their IoT applications on fog, by making the procedure easier, convenient, and with less effort, cost and time wasted.

7.2 Conclusion

In this thesis, I demonstrate how a user can use the implemented workload generator, in order to produce data according to his needs, or replay recorder sensor data, straight to his application, regardless the output protocol or the sensor message formats, since the provided interfaces can be extended to cover the desired scenarios. The provided comprehensive model for describing the IoT devices, along with a scalable implementation of an IoT heterogeneous workload generator framework, can be used for users to experiment real world scenarios and evaluate their applications performance regardless of the application specific needs. The evaluation of the implementation demonstrates the accuracy performance for the exported sensor data, and the performance downgrade factors regarding the sensor message timing accuracy were extracted. It was proved that as bigger is the input sensor number, the greater will be the delay between interval time for sensor messages, and as greater is the declared frequency between sensor messages of the same sensor, the less will be the delay, due to less generated messages. A way to decrease the delay overall, is to use a hosted machine with more available processing resources (more cores), that with the support of parallel execution, minimizes the latency as it was observed and proved. In addition, we found the limitations for our system. A single instance of our system can safely emulate until 8000 sensors in a weak machine (2 cores) and around 30000 sensors in more powerful machines with 16 cores. Finally, we found that the system can emulate and handle up to 4000 sensors in order not to affect the performance for the user application in a machine with 2 cores, and this threshold increases to approximately 6000 sensors in more decent machines with more than 8 cores, for generation rate 2 seconds. Finally, we demonstrated how useful are workload generators in general, and to other systems as well, like edge or fog emulators. The integration of fog emulators with workload generators can provide better analytics and flexibility to the users, and users can extract more insights using this combination when testing their application before deploying it to the real world.

7.3 Future work

There are a lot of enhancements that this work can support, to make it even useful and widely used. The input and output interfaces, encouraging and yelling the implementation of more sensor prototypes, and more output protocols that current IoT devices support (for example CoAP output protocol). Mentioning the IoT devices heterogeneity, more advanced generation rates could be developed, to support more extreme values population methods through generation scenarios. Another enhancement that would make the system even more scalable, is the potential creation of a central control master, in order to support and monitor multiple instances of the workload generator and distribute the workload (or the sensors number) to make it support even more sensors.

Bibliography and References

[1] M. Vega, 2020 [Online]. Available: https://review42.com/internet-of-things-stats

[2] IDC, 2019 [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS45213219

[3] IDC, 2020 [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS46286020

[4] U. S. Shanthamallu, A. Spanias, C. Tepedelenlioglu and M. Stanley, "A brief survey of machine learning methods and their sensor and IoT applications", 2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA), pp. 1-8, Larnaca 2008, doi: 10.1109/IISA.2017.8316459.

[5] S. Madakam, R. Ramaswamy and S. Tripathi, "Internet of Things (IoT): A Literature Review.", Journal of Computer and Communications, vol. 3, no. 5, pp. 164-173, 2015, doi: 10.4236/jcc.2015.35021.

[6] I. Lee, K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises", Business Horizons, vol. 58, issue 4,pp. 431-440, ISSN 0007-6813, 2015, https://doi.org/10.1016/j.bushor.2015.03.008.(http://www.sciencedirect.com/science/article/pii/S0007681315000373)

[7] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications", IEEE Communications Surveys & Tutorials, vol. 17, no. 4, pp. 2347-2376, 2015, doi: 10.1109/COMST.2015.2444095.

[8] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP", IEEE International Systems Engineering Symposium (ISSE), pp. 1-7, Vienna 2017, doi: 10.1109/SysEng.2017.8088251.

[9] R. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, 1999

[10] [Online]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer

[11] U. Hunkeler, H. L. Truong and A. Stanford-Clark, "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks", 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), pp. 791-798, Bangalore 2008, doi: 10.1109/COMSWA.2008.4554519.

[12] ApacheKafka.[Online].Available:http://kafka.apache.org/documentation.html#introduction

[13] A. Sharma, "Apache Kafka: Next Generation Distributed Messaging System", International journal of scientific engineering and technology research, vol. 03, issue 47, pp. 9478-9483, ISSN 2319-8885, Dec. 2014, http://www.infoq.com/articles/apache-kafka

[14] A. Youssef, "Exploring Cloud Computing Services and Applications", Journal of Emerging Trends in Computing and Information Sciences, Jul. 2012

[15] S. Yi, C. Li, and Q. Li, "A Survey of Fog Computing: Concepts, Applications and Issues", In Proceedings of the 2015 Workshop on Mobile Big Data (Mobidata '15), pp. 37–42, New York 2015, https://doi.org/10.1145/2757384.2757397

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things", In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (MCC '12), pp. 13–16, New Work 2012, https://doi.org/10.1145/2342509.2342513

[17] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges", in IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637-646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[18] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. Dikaiakos, “Fogify: A fog computing emulation framework”, in Proceedings of the 5th ACM/IEEE Symposium on Edge Computing, ser. SEC ’20, New York 2020

[19] M. Curiel and A. Pont, "Workload Generators for Web-Based Systems: Characteristics, Current Status, and Challenges", in IEEE Communications Surveys & Tutorials, vol. 20, no. 2, pp. 1526-1546, Secondquarter 2018, doi: 10.1109/COMST.2018.2798641.

[20] N. Bruno and S. Chaudhuri, "Flexible database generators", In VLDB, pp. 1097–1107, 2005.

[21] J. E. Hoag and C. W. Thompson, "A parallel general-purpose synthetic data generator", SIGMOD Record, vol. 36, issue 1, pp.19-24, March 2007, https://doi.org/10.1145/1276301.1276305

[22] T. Rausch, C. Lachner, P. A. Frangoudis, P. Raith, and S. Dustdar, “Synthesizing plausible infrastructure configurations for evaluating edge computing systems”, in 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association, Jun. 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/rausch

[23] O. Kolosov, G. Yadgar, S. Maheshwari and E. Soljanin, “Benchmarking in the dark: On the absence of comprehensive edge datasets”, in 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20). USENIX Association, Jun. 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/kolosov

[24] D. Libes, D. Lechevalier and S. Jain, "Issues in synthetic data generation for advanced manufacturing", 2017 IEEE International Conference on Big Data (Big Data), pp. 1746-1754, Boston 2017, doi: 10.1109/BigData.2017.8258117.

[25] T. D. Doudali, I. Konstantinou and N. Koziris, "Spaten: A spatio-temporal and textual big data generator", 2017 IEEE International Conference on Big Data (Big Data), pp. 3416-3421, Boston 2017, doi: 10.1109/BigData.2017.8258327.

[26] J. W. Anderson, K. E. Kennedy, L. B. Ngo, A. Luckow and A. W. Apon, "Synthetic data generation for the internet of things", 2014 IEEE International Conference on Big Data (Big Data), pp. 171-176, Washington 2014, doi: 10.1109/BigData.2014.7004228.

[27] A. Mäkinen, J. Jiménez and R. Morabito, "ELIoT: Design of an emulated IoT platform", 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), pp. 1-7, Montreal 2017, doi: 10.1109/PIMRC.2017.8292769.

[28] D. Giouroukis, J. Hülsmann, J. V. Bleichert, M. Geldenhuys, T. Stullich, et al.. , "Resense: Transparent Record and Replay of Sensor Data in the Internet of Things", 22nd International Conference on Extending Database Technology (EDBT), Lisbon Mar. 2019. ⟨hal-02056767⟩

[29] D. Mosberger and T. Jin, "Httperf—a tool for measuring web server performance", SIGMETRICS Perform. Eval. Rev. 26, pp. 31–37, 3 Dec. 1998 https://doi.org/10.1145/306225.306235

[30] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis and M. D. Dikaiakos, "StreamSight: A Query-Driven Framework for Streaming Analytics in Edge Computing", 2018 IEEE/ACM 11th

International Conference on Utility and Cloud Computing (UCC), pp. 143-152, Zurich 2018, doi: 10.1109/UCC.2018.00023.

Appendix A

A.1 Repository of Source Code for implementation

The source code can be found in https://github.com/aioann01/IoT-workloadGenerator (personal) or through the UCY LINC LAB at https://github.com/UCY-LINC-LAB/aioannou-ade2020-master-iot-workload-generator , along with sample files, and deployment docker-compose files.