Automated and Reproducible Application Traces Generation for IoT Applications

Nina Santi, Rémy Grünblatt, Brandon Foubert, Aroosa Hameed, John Violos, Aris Leivadeas, Nathalie Mitton

Automated and Reproducible Application Traces Generation for IoT Applications

Nina Santi
Inria Lille
Villeneuve d'Ascq, France
nina.santi@inria.fr

Rémy Grünblatt
Inria Lille
Villeneuve d'Ascq, France
remy.grunblatt@inria.fr

Brandon Foubert
Inria Lille
Villeneuve d'Ascq, France
brandon.foubert@inria.fr

Aroosa Hameed
École de technologie supérieure
Montréal, Canada
aroosa.hameed.1@ens.etsmtl.ca

John Violos
École de technologie supérieure
Montréal, Canada
violos@mail.ntua.gr

Aris Leivadeas
École de technologie supérieure
Montréal, Canada
aris.leivadeas@etsmtl.ca

Nathalie Mitton
Inria Lille
Villeneuve d'Ascq, France
nathalie.mitton@inria.fr

ABSTRACT

In this paper, we investigate and present how to generate application traces of IoT (*Internet of Things*) Applications in an automated, repeatable and reproducible manner. By using the FIT IoT-Lab large scale testbed and relying on state-of-the-art software engineering techniques, we are able to produce, collect and share *artifacts* and datasets in an automated way. This makes it easy to track the impact of software updates or changes in the radio environment both on a small scale, e.g. during a single day, and on a large scale, e.g. during several weeks. By providing both the source code for the trace generation as well as the resulting datasets, we hope to reduce the learning curve to develop such applications and encourage reusability as well as pave the way for the replication of our results. While we focus in this work on IoT networks, we believe such an approach could be of used in many other networking domains.

CCS CONCEPTS

• **Networks** → **Network experimentation**; *Ad hoc networks*; • **Computer systems organization** → *Sensor networks.*

KEYWORDS

datasets, testbed, network, traces, reproducibility, automation, experiments, 802.15.4

ACM Reference Format:
Nina Santi, Rémy Grünblatt, Brandon Foubert, Aroosa Hameed, John Violos, Aris Leivadeas, and Nathalie Mitton. 2021. Automated and Reproducible Application Traces Generation for IoT Applications. In *Proceedings of the 17th ACM Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet '21), November 22–26, 2021, Alicante, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3479242.3487321

1 INTRODUCTION

Data is now at the heart of many research. With the recent growth of machine learning, in particular deep learning, it is now a critical task to find exhaustive and quality data sets to produce pertinent and realistic models. It is even more the case in the IoT, where network traffic data is essential to understand the network characteristics, model them and their patterns. Researches use this kind of data for training models to identify and classify the different IoT devices present in the network. It helps administrators manage the mass of devices in the network and detect malicious devices [25, 29]. It is even more important to train models on traffic data because the IoT devices identification is complicated and relies on easily changeable static information and not on their less changeable activity. Network data sets are also useful to detect anomalous activity or signature behavior of already known attacks by training machine learning [18, 32] or deep learning [12] models. Finally, network IoT data alleviates forecasting networks congestion [30] or the traffic load [23] to tackle allocation and management decisions.

Because various types of bias may be present in data [24], and such bias may in turn lead to algorithmic *unfairness* or other unwanted behaviours, it is important to take great care when designing and creating datasets. Although replicability[1] appears as the best way to increase confidence of results building on such experimental artifacts, intermediate steps such as repeatability and reproducibility may also promote the integrity of such results. Repeatability for the creation of datasets is especially important for the IoT and wireless networking domains, as the radio spectrum, the communication technologies or the operating systems powering such networks undergo constant changes, rendering datasets obsolete, or worse, damaging.

In this work, we use the FIT IoT-LAB testbed. FIT IoT-LAB is an open large-scale testbed with wireless sensor nodes and mobile robots [1]. Each node is openly programmable and it comes with convenient tools to gather the experiments' results, making it ideal to generate datasets. We show how we generated different types of network traffic data with the FIT IoT-LAB testbed thanks to our Sisyphe tool. In addition, we provide our results, methods and

[1]We use the terminology of the Artifact Review and Badging Version 1.1 from the ACM.

tools to be entirely reusable and reproducible and share them. Our contributions are threefold :

- We present the architecture of Sisyphe and illustrate its use by generating two datasets using the FIT IoT-Lab testbed.
- For each dataset, we discuss the experiment setup and present the results obtained. These results are available to be directly usable.
- We make available all the source code and tools used for each dataset on a public repository.

The remainder of this paper is organized as follows. Section 2 presents the architecture of Sisyphe . Section 3 introduces the two different considered scenarios and their peculiarities. The scenarios are then covered in details in 3.1 and 3.2. Section 4 provides some usage examples of these datasets. Finally, Section 5 positions Sisyphe and these contributions with regards to literature before Section 6 concludes this paper by discussing future work.

2 SISYPHE DESIGN PRINCIPLES AND ARCHITECTURE

The main goal of Sisyphe is to provide a way to generate artifacts in an autonomous, repeatable and reproducible manner[2]. To achieve this goal, Sisyphe relies on multiple techniques or design principles that we present in this section. We then present its architecture.

2.1 Design principles

It is notoriously hard to reproduce or let alone repeat computer systems research. In [9], the authors measure the two most "basic" requirements for reproduction (different teams, same experimental setup), namely, being able to access to the source code and being able to build this source code, for a set of 601 papers from various ACM conference and journals. Out of the 402 papers not excluded from consideration for hardware availability constraints, or excluded because their results were not backed by code, only 54.0% fulfill the two basic requirements of source code availability and build success. While hopefully the situation is far less dire in 2021 than in 2015, the first step to guaranty reproducibility and repeatability is to ensure the availability of the source code involved in the processing, analysis, presentation steps of the creation of scientific works, but also and most importantly of the *experimental code.*

Source code archiving. To tackle the source code availability problematic, the source code of the Sisyphe platform as well as the source code of the experiments run on the platform are distributed under open-source licences. Yet, distributing the source code, for example by hosting an archive on some institutional or personal web-page is not enough. Indeed, they may disappear on the occasion of a website redesign or a change of affiliation. Thus, source code sharing platforms such as Github or Gitlab represent a more promising avenue. However, recent history [5, 7, 19] has shown they are not perfect because they can also close. To ensure source code availability, we therefore rely on software archiving made possible by Software Heritage [10]. It is a non-profit organization backed by Inria and the UNESCO, whose goal is to establish a long-term software and source code archive. In particular, unique and stable identifiers called SWHID are attributed to the different software components of Sisyphe ensuring an easy access to those components source code.

For example, the identifier of the Sisyphe source code, available at https://github.com/sisyphe-re/infrastructure-ng is

`swh:1:dir:6b23bcefe73bbaea308b6ea99d7e75cd37ded16fe`

and allows the exploration of a project's source code in a similar way to the Github web interface. In addition to preserving the source code of the platform, Software Heritage is also used to archive the source code of the *experiments* every time they are run in an automated manner. In particular, we ensure that the source code is present in the Software Heritage archive before running the experiments, preventing mismatch between the ongoing experiment and what is archived in a long-term manner.

Reproducible Builds. Any researcher that already tried to run old code they wrote for a scientific contribution knows how hard this can be. Having access to the source code is only one part from the broader "reproducibility challenge", as underlined by the results from the "Ten Years Reproducibility Challenge" presented in [27] : lacking documentation or access to obsolete computing environments can also represent hard obstacles to overcome. In Sisyphe , we use the Nix package manager [14] and the NixOS [15] Linux distribution to provide reproducible development and computing environments. Nix uses a functional approach to package management by linking software and their dependencies using cryptographic hashes. It so allow us to provide an easy way for third-party to re-use our code, without impacting the rest of their system. For example, it is possible to build the RIOT-OS based firmwares from Section 3.2, without having to deal with the so-called "dependency hell", and as long as the Nix package manager is available, by using a single command, namely ‘nix build‘, as shown on 1.

Listing 1: Firmware Compilation using Nix

```
$ git clone git@github.com:sisyphe-re/firmwares
$ cd firmwares/
$ nix build                                    [1 built]
$ ls result/
gnrc_border_router.elf   gnrc_networking.elf
```

The more global architecture of Sisyphe is itself written in Nix and based on the NixOS Operating System and thus adopt the "Infrastructure As Code" paradigm. Nix and NixOS try to be reproducible down to the bit level, they so detect quickly existing projects that exhibit non-deterministic patterns in their builds. We discovered an example of such behaviour in the RIOT operating system's code during the development of Sisyphe [3].

Testbeds. As mentioned in [9, 27], hardware access may also hinder reproducibility, especially when it involves *exotic* hardware. Providing remote access to standard x86_64 machines might not be useful for reproducibility of system research, as it is easy to gain access to such machines (at least in 2021), but networks and especially IoT networks are a different matter. The fast pace of evolution of such networks means some hardware can be sunset in just a few years time. Moreover, their diversity means reproducing some experiment might come at a high cost, in terms of hardware roll-out or time. We ease the reproduction process in Sisyphe by

[2] *Sisyphe* is the french name of the Greek mythology character *Sisyphus*, punished for cheating death by being forced to *repeat* an action for eternity.

[3] https://github.com/RIOT-OS/RIOT/pull/16511/

using testbeds, instead of relying on local deployments. By doing so, anybody with an access to the testbed can reproduce the results at a lower cost. In addition, testbeds promote a greater access to science by leveraging more resources for more people, in particular in IoT networks that demand large deployments. In this work, we use the FIT IoT-Lab testbed, which allows full control of several hundred nodes using a set of APIs and SSH access across five geographical deployments in France. One of the downside of such remote-accessible testbeds, as noted by [4] is that they offer rigid operating conditions and might not provide meaningful results because of their peculiar deployments that may be considered as *artificial.* To this end, SISYPHE is testbed agnostic and does not try to enforce the use of any specific testbed, as long as experiments can be run in an automated manner, which brings the next main design paradigm of SISYPHE .

Continuous Integration and Automation. The radio environment evolves quickly with the evolution of radio spectrum allocation, changes in the users habits, changes in the communication technology, changes in the applications, operating systems. Thus, SISYPHE adopts a *continuous integration* and full *automation* approach for the creation of datasets. Such automation allow identification of IoT networks' global trends, for example about the performance of this or that algorithm implemented in some operating system. Thus, it gain insight into the development of real networks. Most importantly, full automation decouples the experiment from the experimenter and highlight implicit dependencies that may otherwise be hidden. Indeed, the initial work put into automation effort is a form of documentation. It allows observability of the whole experimental process and thus reduce the risk of *corporate amnesia* and favor easier audits.

2.2 Architecture

The global architecture of SISYPHE is presented in Figure 1. The processing pipeline is divided in two main steps, each with different level of reproducibility. The first one is the *automated build* step, which uses experiment blueprints ,from source code repositories, to create artifacts. These artifacts are then used during the second step, which is the *automated evaluation* step.

SISYPHE built blueprints with Nix flakes, which are experimental features for easy containerization and composability of Nix-based projects [13]. This assumes that the blueprints are already nix and flakes *ready*, like in Section 3.2. In the case they are not, a bit of work might be needed as shown in Section 3.1 and some intermediate *adapter* code is needed. SISYPHE assumes that calling "`nix build`" will build the artifacts and expose a "`run`" executable for launching the experiment. The build results are published online in a binary cache in which they are identified by their hashes, which allows to fetch part of the artifacts without having to rebuild them locally.

The automated evaluation step is in general *less* reproducible than the automated build step. Indeed, for the evaluation of experiment we have to make API calls and interact with real-world resources, which is not as simple as running a sequence of programs in a controlled environment. This evaluation step is executed automatically on SISYPHE at regular intervals, e.g. daily, and the result are also made available on the web[4]. We may need secrets to connect to platforms such as Fit IoT-LAB, for example ssh private keys to copy artifacts on SISYPHE server for their availability, or API keys. Those are passed as environment variables. Containers are used to separate each experiment's evaluation, but we plan to switch to virtual machines in the near future to increase security.

3 USAGE EXAMPLE : IOT ARTIFACTS GENERATION

In this section, we present how we have produced two IoT artifacts with SISYPHE on the FIT IoT-Lab platform. The two artifacts are created using scenarios related to *smart buildings* [26, 31], covering *(i)* HVAC systems (heating, ventilation and air conditioning), *ii)* Smart lighting, *iii)* Emergency system, *iv)* Surveillance cameras, *v)* Virtual or augmented reality rooms and *vi)* Voice over Internet (VoIP). Each sub-scenario is characterised by the frequency of the exchanged messages, the number of nodes participating in the network, as well as the payload size. With these scenarios, we aim to model the exchange of different types of data, whether textual, audio, or visual data.

The first artifact uses the Contiki [16] IOT Operating System, and is rather loosely interfaced with SISYPHE . Indeed, while the compilation environment is set up using Nix in SISYPHE , Nix is not used directly to build the firmwares. The source code is fetched *on-the-fly* in a non-reproducible manner over the internet, that is to say using Git. Any change in the source code repository will not be detected by the system: this first evaluation scenario is used to present how an existing code could be used in SISYPHE with minimal changes. The duration of the experiments are limited to at most 90 minutes. The second artifact uses the RIOT IOT Operating System [2], and is SISYPHE -*ready*, as it already uses Nix to build the firmwares and the different scripts used for the experiment execution. This second artifact is used to present how a more *tailored* experiment code can be used with SISYPHE . The duration of the experiments are longer, as each scenario runs for one week. Both methods rely on sensors writing statistics on their serial output to gather data, serial output which is connected to the FIT IoT-Lab monitoring infrastructure and gathered with the so-called "serial-aggregator" of the testbed, and on the use of the 802.15.4 radio technology.

3.1 Simple traces for link evaluation

Based on the needs and requirements from a specific usage, we may need to use simple data about node communication in a peer to peer approach. It may for example be useful to train machine learning models to get an estimation of the communication delay. In this section, we first present the experimental setup and tools we used to configure and launch experiments on the FIT IoT-LAB test-bed. Then we introduce the tools we used to retrieve the experimental results and parse the files to get data about specific metrics we needed. The methodology presented can be applied to any other metric that can be extracted from a set of nodes communicating in broadcast manner.

3.1.1 Experiment setup. FIT IoT-LAB offers several tools to launch, control and retrieve results from the experiments. The nodes' firmware

[4]The web interface for SISYPHE is available at https://sisyphe.grunblatt.org/

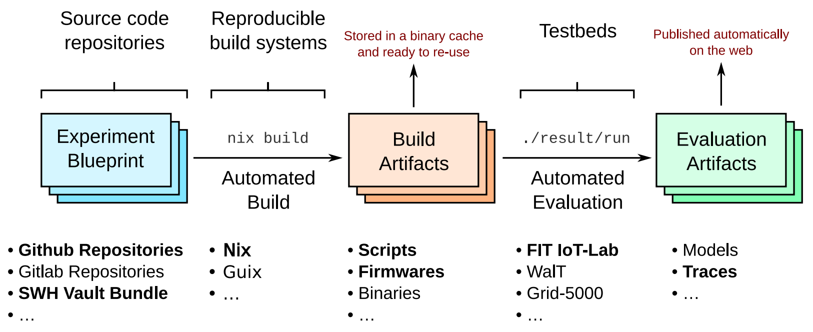

Figure 1: Overview of the architecture of Sisyphe . Bold elements are used in one of the two example presented in this work.

can be fully modified, and several examples are available to learn. Different operating systems can be used, such as Contiki [16], which is a lightweight system for wireless sensor networks. To generate simple traces for link evaluation, we based our firmware on a publicly available example based on Contiki [5] which sends broadcast messages on a regular basis. We made small modifications to adjust the behavior to our needs. To easily and quickly run experiments, we wrote a simple script which takes the following experimental parameters as input: the experiment duration, the sending frequency, the packets' payload size and the list of nodes used in the experiment. We have defined these parameters according to the characterization of the scenario done previously. They are summarized in Table 1. To get information about the nodes operations, they are configured to output data on their serial links as soon as they send or receive packets. For the hardware side, we chose to use the open nodes M3 [6]. Those are based on STM32 micro-controllers and an ARM Cortex M3 processor. They offer power and flexibility, and thus are the ones that meets the largest number of applications. Our script execute the following operations. The firmware code is modified, based on the experimental parameters that are given. Then the firmware is compiled, and the nodes that takes part in the experiment are booked using the IoT-LAB tools. The serial aggregator from the toolset is started to log the serial output of every node from the experiment. For every line of log, a timestamp is registered along the source node's name. Once the experiment is ready to begin, the firmware is uploaded on the nodes. After the given duration of the experiment, the results are retrieved from the remote IoT-LAB server and a copy is stored on the local computer. The script and every software we used are freely available and usable with the obtained data.

3.1.2 Results. Thanks to the serial output generated by the nodes, we can extract simple experiments' results. A single text file contains output from every nodes. Each of that file's line contains information about which nodes sent messages, and which nodes received it. A line of log contains a timestamp, the identifier of the node which wrote the line, if it has sent or received a message, and the associated message identifier. As the log structure is very simple, the file's parsing is easy. We divide the log file in two : one contains the log about messages sent and the other log about messages received. In the first file, about messages sent, we compute and add the number of nodes that have received each message next to the log line of its transmission. In the second file about receptions, based on the timestamps of each log, we compute and add the transmission delay in milliseconds associated with each reception. With that, we obtain a simple yet complete dataset about one hop communications. That dataset can then be used to train automatic learning models which needs realistic information and metrics about specific applications and experimental settings. In our work, we use the dataset to predict the communication's delay, based on the network's state and use case. To that end, we wrote scripts to automatically extract, compute, and graph different metrics about the dataset. Those scripts are made with Python, and freely available in open access on our Gitlab repository[7]. Based on the needs, metrics can be simply extracted and graphed with quick modifications of the available scripts. Our scripts are intended to be reusable to extract other metrics, such as the message window, or other data with the same format.

3.2 Traces for network evaluation

In this section, we focus on creating a more realistic dataset for network evaluation. This type of dataset is useful to understand applications traffic profiles. We first describe the experiment architecture and setup. We also introduce how we generate the traffic and the experiment parameters. Then, we present the results we obtained and how to exploit them.

3.2.1 Experiment description. We reuse the scenarios of the Section 3 but enhance them by considering different type of packet generation. As we focus on network evaluation, we present a more advanced architecture, illustrated in Figure 2. Each experiment, corresponding to a scenario, involves three types of entities (nodes):
Server: the server is a node which hosts a UDP server, receiving packets and collecting information about those packets. It is reachable through the Internet through an IPv6 connection. In all the experiments, there is only one server.
Border Routers : border routers are nodes which connect the sensors to the internet. They are equipped with two interfaces, one connected to the internet and the other one connected to the sensors network, using the 802.15.4 technology and the RPL routing protocol. In the RPL protocol, they are the root of the RPL DODAGs.

[5]Available here: https://github.com/iot-lab/contiki/tree/master/examples/ipv6/simple-udp-rpl
[6]More information: https://github.com/iot-lab/iot-lab/wiki/Hardware-M3-node
[7]https://gitlab.irisa.fr/0000H82G/traces

Table 1: Experiments parameters for simple link evaluation

Scenario	Nodes number	Message exchange frequency	Payload size (byte)	Duration (minutes)
HVAC systems	100	1 packet/4 minutes	60	60
Smart lighting	100	1 packet/8 minutes	30	90
Emergency response	40	1 packet/30 seconds	127	10
Surveillance	30	99 packets/seconds	127	10
Virtual or augmented reality	10	197 packets/seconds	127	10
VoIP	10	16 packets/seconds	127	10

They are at the edge of the sensor network, bridging two networks. During the experiments, the number of border routers is kept constant but may vary between experiments, as shown in Table 2.
Sensors : The sensors are the core of the network. They generate data according to a specified random distribution and transmission parameters, and send them to the server. They are connected together using the 802.15.4 technology and the RPL routing protocol, which means each sensor may also be used to relay packets or frames to border routers, if it is on the shortest path between a sensor and the DODAG root. Each sensor may have different parents in the DODAG, allowing packets to flow in a path or another.

We introduce a new type of packet generation which goes beyond regular and periodic packet generation, and allows to better model real world scenarios. Indeed, in the HVAC scenario for example, sensors might send information in a periodic manner in order to control the system, but in the emergency response scenario for example, sensors will also react when they detect a possible danger, which cannot be captured by simple periodic traffic generation. Thus, beyond the previously used periodic generation where the packet are sent at a fixed interval of time (denoted as "Periodic"), we also use a packet generation method based on an exponential distribution of rate parameter λ (denoted as "Exponential"). This distribution is generated by using the inverse transform sampling method directly on the sensor node, using the floating point random number generation from the RIOT Operating System. As the FIT IoT-Lab m3 nodes are based on the STM32 CPUs, which feature a hardware random number generator, we believe the generated distribution is of sufficient quality for our usecases. We also consider an *Hybrid* traffic generation method, which is the simultaneous generation of data according to the *Periodic* and the *Exponential* method at the same time. All parameters are summarized in Table 2.

3.2.2 Results. The results are gathered either from the server or from the sensors by a script running on the FIT IoT-Lab frontend servers. In particular, the border routers are not gathering any statistics to avoid such data collection to interfere with the experiment, as their serial link is already for the ethernet-over-serial (ethos) connection. The results for each scenario are streamed and compressed using the ZSTD compression algorithm[8] directly to Sisyphe server where they can be post-processed after the end of the experiment, in particular to format them. Originally formatted as text, the results are transformed into SQLite3 databases to allow for easy manipulation and querying. For the moment, we kept the database format simple, in particular by using no relationship between the tables, using SQLite3 as a poor man's binary file format. Such databases are also compressed using ZSTD, as their uncompressed size can reach 100GB for an experiment of one week.

The databases comprise tables with different level of information, from high level layer 3 statistics to low level phy information, e.g. average reception power of the frames of this or that neighbor in the RPL DODAG. While we will not go into the details of the database schema, which is documented in the campaign repository [9], we present in Listing 2 an example of operation on the data. Theses operations give us the transmission timestamp, the sensor ID and reception timestamp for the first 5 received packets of the dataset.

Listing 2: Example of simple data manipulation of the generated data

```
$ sqlite3 hvac_263685.db3
SQLite version 3.35.2 2021-03-17 19:07:21
Enter ".help" for usage hints.
sqlite> SELECT udp.Timestamp, udp.Node, server.Timestamp FROM server
   ...> INNER JOIN udp ON server.payload=udp.payload LIMIT 5;
2021-05-06 16:34:22.145909070|m3-377|2021-05-06 16:34:22.170198
2021-05-06 17:09:02.288019895|m3-367|2021-05-06 17:09:02.308101
2021-05-06 17:22:02.258351087|m3-379|2021-05-06 17:22:02.296746
2021-05-06 18:31:22.448118925|m3-42|2021-05-06 18:31:22.461107
2021-05-06 18:57:22.298322916|m3-362|2021-05-06 18:57:22.324690
```

Such data can for example be presented as in Figure 3, where we plot the latency of every received frame in the *HVAC systems* scenario for each received frame at the server side. Out of the 128 627 received frames, 115 have a *negative* latency (and are not represented on the graph), i.e. they appear as if they were received before being sent. We believe this may be due to the FIT IoT-Lab serial_aggregator behaviour which is used to time the received serial messages from the sensors, or due to some latency introduced by the RIOT threading mechanism, both of which may introduce some latency before *recording* that a packet has been sent. We can observe that as expected, the DODAG roots, whose transmitted packets are represented in yellow, have an overall lower latency as they are closer to the server. We present in Figure 4 the packet delivery ratio over the whole duration of the *HVAC systems* experiment for each node (with each point representing a single node). We can observe that out of the 100 sensors, 34 have a PDR of less than 1.2%, including 15 nodes never having a single packet received by the

[8]https://github.com/facebook/zstd

[9]https://github.com/sisyphe-re/riot_rpl_udp_scenarios

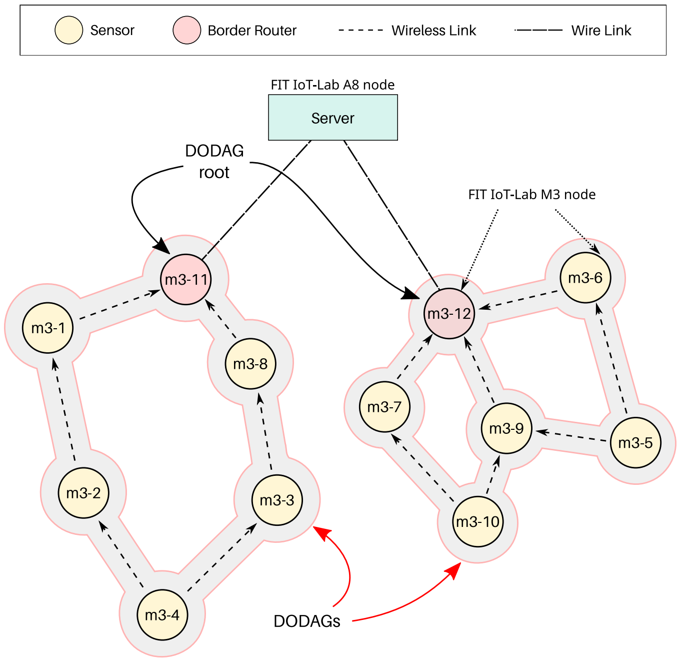

Figure 2: Experiment architecture for network evaluation

server. We attribute this behaviour to the physical topology of the Grenoble FIT IoT-Lab testbed where M3 nodes may be too far to communicate even if their IDs are continuous. A smarter selection of the nodes should therefore be considered for future experiments.

4 POSSIBLE USAGES

Traces of IoT applications are valuable information for wireless networks architectures, like edge computing. We can use datasets to train and evaluate resource allocation and task offloading mechanisms. Edge computing is characterized by a highly dynamic environment and heterogeneity of applications and resources. Thus, data driven modeling is a prominent solution for this type of network. Real time schedulers can recognise trace patterns and allocate resources such as memory, bandwidth and CPU, among users to optimize the network efficiency and reduce costs. Scheduling algorithms [8] decide the amount of resources allocated and the duration of each allocation into two different layers. In the first layer, resources from servers are distributed among several virtual machines. In the second, a virtual machine can run several applications concurrently. The complexity of edge computing systems, the fluctuation of the workload, the constraints of the Service Level Agreements (SLA), the interaction with application's users and the dynamic environment make us move towards intelligent solutions that learn from historical data.

Resource management ensure the smooth operation of the IoT applications and the edge infrastructure. We can train resource management policies with IoT application traces. We can design an admission control to prevent the processing nodes from accepting workload that will cause QoS degradation, such as an increase in response time or a decrease in throughput. An efficient resource allocation mechanism increase the computational efficiency of the infrastructure. It consider the amount of resources required for the timely completion of the workload, the vicinity between the workload producers to the processing nodes [33] and the type of workload [3]. For instance, an intelligent resource allocation mechanism can offload to GPUs the workload produced by augmented reality IoT applications because this type of workload includes intensive matrix multiplications. Last but not least, a load balancing mechanism avoids the situation in which some nodes become overloaded while the others have little work to do. The workload balancing mechanism can redistribute the workload evenly among processing edge nodes, trying to ensure the allocation principles of efficiency, fairness and starvation-free of the nodes.

A more challenging task is the workload modeling. Workload is characterized by self-similarity with multivariate dependencies, strong auto-correlation values and seasonality [6]. With a time series analysis of IoT application traces, we can detect changes in the evolution of the workload, find relationships between different steps in time and forecast future values. We can analyze historical data sequences to estimate the correlations between different edge nodes and predict bottlenecks in the network. The workload modeling may include two components. The first one build IoT applications profiles based on : the message exchange frequency, payload, transmission time, packet delivery rate, resource usage. These applications profiles will summarise how an application behaves in different circumstances. The second component associate application profiles with resource requirements or QoS metrics. The workload modeling will help the application owners and the edge infrastructure providers to understand what are the main characteristics of the application, how the applications interact with the infrastructure and which nodes are identified as hubs or bridges.

The extensive use of IoT devices and edge infrastructures leads to service reliability and availability issues. These issues are intrinsic to edge computing because of its highly distributed nature. Several types of failures may occur such as hardware, virtual machines, application and network. It may cause full or partial breakdowns and shutdowns. Edge computing should be resilient to these failures. Conventional replication techniques may not be feasible because on the edge resources are not highly available. Analysing IoT applications traces can help us to detect and handle the various faults with two fault tolerance approaches: proactive and reactive [17]. In proactive fault-tolerance, the data traces can be used to predict potential faults and substitute the suspected node by some replicant node. In the reactive fault tolerance, we use data traces to detect the fault and take actions in order to decrease the influence of failure to improve the recovery time and the maintainability of the system.

We require an advanced analysis to get meaningful insights of IoT QoS applications' behavior from generated time series datasets. QoS requirements, such as throughput, delay, and Packet Loss Ratio (PLR), associated with each IoT application helps us understand the main characteristics of these applications. Furthermore, there is a large number of IoT devices generating huge amounts of data within each application. Thus, if we predict the time varying characteristics of the device's traffic, we can guarantee a specific level of Quality of Service (QoS). These QoS can be quantitative or ratio data type, representing the application datasets. The prediction of such metrics, by resorting to machine learning approaches, becomes of utmost importance. The machine learning architectures

Table 2: Experiments parameters for network evaluation

Scenario	Sensor number	Border routers number	Payload size (bytes)	Packet generation type	lambda	Period (s)
HVAC systems	100	5	60	Periodic	-	260.0
Smart lighting	100	5	30	Exponential	0.00208	-
Emergency response	40	5	127	Hybrid	0.0333	30.0
Virtual or augmented reality	10	3	127	Exponential	196.74	-
VoIP	10	3	127	Hybrid	15.74	.063532

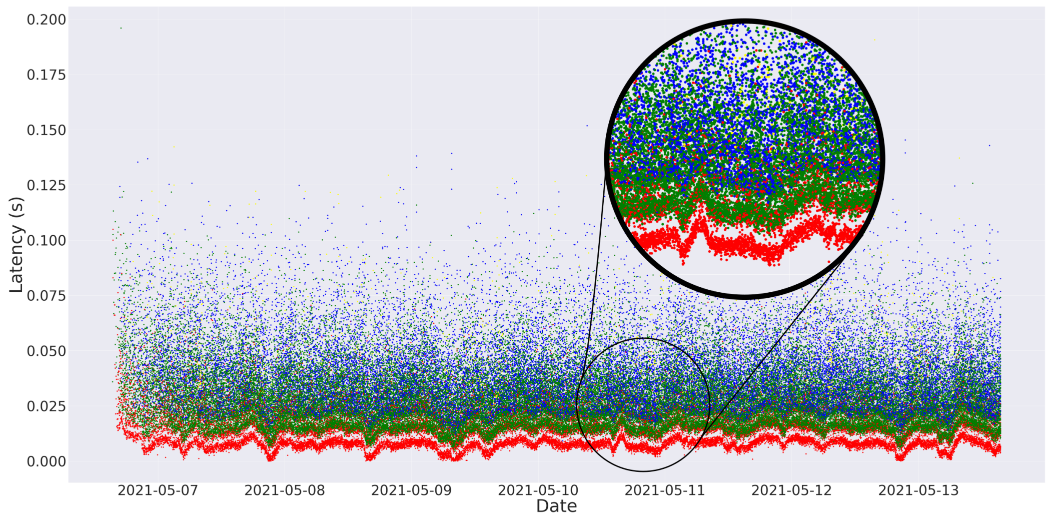

Figure 3: Evolution of the latency for the network evaluation HVAC scenario as a function of time. Each dot represents a single received packet, and colors represent the Rank in the RPL protocol, with red being 256, green being 512, blue being 1024 and yellow above.

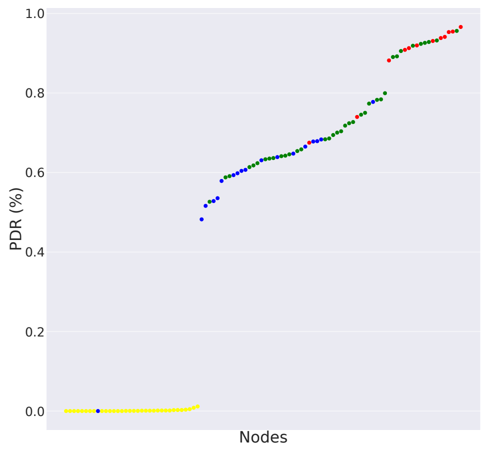

Figure 4: Packet Delivery Ratio (PDR) for the 100 nodes from the network evaluation HVAC scenario over the whole experiment duration. The colors represent the mode of the RPL Rank using the same color code from Figure 3.

efficiently extract useful features and suppress irrelevant variations in datasets [22]. It has an ability to approximate the complex functions in order to build the prediction models for learning tasks and desired accuracy. Thus, machine learning constructs a powerful tool for QoS prediction problems by leveraging generated IoT application traces. We can predict the QoS metrics with historical datasets with various learning models, such as Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM). In a supervised learning setting, the relation between a set of input variables and one or more output variables is determined using a finite set of observations. Once the machine learning model is trained then the prediction is done. For example, statistical based features (i.e. packet inter-arrival time, burstiness rate, etc.) and network based features (i.e. IP addresses, port numbers, packet sizes, etc.) can be employed to predict the individual's IoT device traffic profile [?]. Finally prediction of the IoT traffic generated is done to better understand services' requirements.

5 RELATED WORKS

Pflanzner et al. [28] provide an indexing service to retrieve open traces of applications available on the internet. [21] presents a dataset from a test bench of solar-powered weather sensors. The data contains the power consumption and production of the application to help research in self-powered IoT applications. Xiao and al. [34] introduce a ready-to-use on-board unit solution to collect vehicle trajectory data. Their solution contains a deep learning algorithm to fill gaps in data collection, such as GPS failures. [20] provides a dataset from the FlockLab platform. Data collection is performed over a long period of time to capture long-term periodic patterns that affect the quality of wireless links. Software Heritage is at the heart of the source code archiving part of Sisyphe [11], but is not aimed at archiving experiment results or binary artifacts (such as firmwares) nor providing the automation for experiments, from the build to the results recovery, unlike Sisyphe .

6 CONCLUSION

In this work, we introduce Sisyphe for an automated generation of *artifacts*. Its architecture automates the build of the various tools needed for experimental evaluation on platforms such as FIT IoT-Lab as well as the artifacts generation, resulting in an entirely repeatable and reproducible process. Experimenters using Sisyphe can easily repeat their experiments according to custom schedules, allowing to account for example for the time variability of environmental factors such as network traffic. Results obtained through Sisyphe can be easily promoted as they are more easily verifiable by their peers. We then illustrate the use of Sisyphe by generating two IoT data sets using the FIT-IoT Lab platform. The presented *artifacts* are from IoT applications, but we emphasize that Sisyphe can generate *artifacts* for other type of networks, depending on the chosen build process and evaluation platforms in the Sisyphe pipeline. Finally, we provide an extensive review of the possible use of this type of data. All code source and tools are at the disposition of the community. We hope with this work to make traces generation accessible along with promoting open science and research reproducibility.

ACKNOWLEDGMENTS

This work was partially supported by the CPER DATA and CHIST-ERA-2018-DRUID-NET project "Edge Computing Resource Allocation for Dynamic Networks".

REFERENCES

[1] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, and J. Vandaele. 2015. FIT IoT-LAB: A large scale open experimental IoT testbed. In *IEEE World Forum Internet of Things (WF-IoT)*.

[2] E. Baccelli, O. Hahm, M.s Günes, M.and Wählisch, and T. C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *IEEE conference on computer communications workshops (INFOCOM WKSHPS)*.

[3] I. Bouras, F. Aisopos, J. Violos, G. Kousiouris, A. Psychas, T. Varvarigou, G. Xydas, D. Charilas, and Y. Stavroulas. 2019. *Mapping of Quality of Service Requirements to Resource Demands for IaaS.* Pages: 270.

[4] P. Brunisholz, E. Duble, F. Rousseau, and A. Duda. 2016. WalT: A Reproducible Testbed for Reproducible Network Experiments. In *IEEE INFOCOM Workshop on Computer and Networking Experimental Research Using Testbeds (CNERT)*.

[5] A. Buchholtz-Au. [n.d.]. *Visual Studio Codespaces is consolidating into GitHub Codespaces.* https://devblogs.microsoft.com/visualstudio/visual-studio-codespaces-is-consolidating-into-github-codespaces/

[6] M.C. Calzarossa, L. Massari, and D. Tessera. 2016. Workload Characterization: A Survey Revisited. *Comput. Surveys* 48, 3 (Feb. 2016), 48:1–48:43.

[7] Denise Chan. [n.d.]. *Sunsetting Mercurial support in Bitbucket - Bitbucket.* https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket

[8] L. Cherkasova, D. Gupta, and A. Vahdat. 2007. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review* 35, 2 (Sept. 2007), 42–51.

[9] C. Collberg, T. Proebsting, and A.M. Warren. 2015. Repeatability and benefaction in computer systems research. *University of Arizona TR* 14, 4 (2015).

[10] R. Di Cosmo and S. Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *(iPRES) Int. Conference on Digital*.

[11] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 2019. Referencing Source Code Artifacts: a Separate Concern in Software Citation. *Computing in Science and Engineering* (Dec. 2019), 1–9. https://doi.org/10.1109/MCSE.2019.2963148

[12] A. Diro and Naveen C. 2018. Distributed attack detection scheme using deep learning approach for Internet of Things. *Future Generation Computer Systems* 82 (2018), 761–768.

[13] E. Dolstra. [n.d.]. *Nix Flakes, Part 1: An introduction and tutorial - Tweag.* https://www.tweag.io/blog/2020-05-25-flakes/

[14] E. Dolstra. 2006. *The purely functional software deployment model.* Utrecht University.

[15] E. Dolstra and A. Löh. 2008. NixOS: A purely functional Linux distribution. In *13th ACM SIGPLAN international conference on Functional programming*.

[16] A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*.

[17] M. Hasan and MS Goraya. 2018. Fault tolerance in cloud computing environment: A systematic survey. *Computers in Industry* 99 (Aug. 2018), 156–172.

[18] E. Hodo, X. Bellekens, A. Hamilton, PL Dubouilh, E. Iorkyase, C. Tachtatzis, and R. Atkinson. 2016. Threat analysis of IoT networks using artificial neural network intrusion detection system. In *Int. Symposium on Networks, Computers and Communications (ISNCC)*.

[19] M. Imbert. [n.d.]. *InriaForge: Forum: Annoucement: Scheduled INRIA Forge end of life.* https://gforge.inria.fr/forum/forum.php?forum_id=11543

[20] R. Jacob, R. Da Forno, R. Trüb, A. Biri, and L. Thiele. 2019. Dataset: Wireless Link Quality Estimation on FlockLab - and Beyond. In *Proceedings of the 2nd Workshop on Data Acquisition To Analysis*.

[21] M. Kuzman, X. d. Toro García, S. Escolar, A. Caruso, S. Chessa, and J. C. López. 2019. A Testbed and an Experimental Public Dataset for Energy-Harvested IoT Solutions. In *IEEE Int. Conference on Industrial Informatics (INDIN)*.

[22] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539

[23] M. Lopez-Martin, B. Carro, and A. Sanchez-Esguevillas. 2019. Neural network architecture based on gradient boosting for IoT traffic prediction. *Future Generation Computer Systems* 100 (2019), 656–673.

[24] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan. 2019. A Survey on Bias and Fairness in Machine Learning. *CoRR* (2019).

[25] M. Miettinen, S. Marchal, I. Hafeez, T. Frassetto, N. Asokan, A. Sadeghi, and S. Tarkoma. 2017. IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*.

[26] D. Minoli, K. Sohraby, and B. Occhiogrosso. 2017. IoT Considerations, Requirements, and Architectures for Smart Buildings—Energy Optimization and Next-Generation Building Management Systems. *IEEE Internet of Things Journal* 4, 1 (2017), 269–283. https://doi.org/10.1109/JIOT.2017.2647881

[27] J. M Perkel. 2020. Challenge to scientists: does your ten-year-old code still run? *Nature* 584, 7822 (2020).

[28] T. Pflanzner, Z. Feher, and A. Kertesz. 2019. A Crawling Approach to Facilitate Open IoT Data Archiving and Reuse. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*.

[29] A. Sivanathan, H.H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman. 2019. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing* 18, 8 (2019), 1745–1759.

[30] F. Tang, Z. Md. Fadlullah, B. Mao, and N. Kato. 2018. An Intelligent Traffic Load Prediction-Based Adaptive Channel Assignment Algorithm in SDN-IoT: A Deep Learning Approach. *IEEE Internet of Things Journal* 5, 6 (2018), 5141–5154.

[31] A. Verma, S. Prakash, V. Srivastava, A. Kumar, and S. C. Mukhopadhyay. 2019. Sensing, Controlling, and IoT Infrastructure in Smart Building: A Review. *IEEE Sensors Journal* 19, 20 (2019), 9036–9046. https://doi.org/10.1109/JSEN.2019.2922409

[32] A. Verma and V. Ranga. 2020. Machine Learning Based Intrusion Detection Systems for IoT Applications. *Wireless Personal Communications* 111 (2020), 2287–2310.

[33] J. Violos, E. Psomakelis, K. Tserpes, F. Aisopos, and T. Varvarigou. 2019. Leveraging User Mobility and Mobile App Services Behavior for Optimal Edge Resource Utilization. In *International Conference on Omni-Layer Intelligent Systems (COINS '19)*. Association for Computing Machinery, New York, NY, USA.

[34] Z. Xiao, F. Li, R. Wu, H. Jiang, Y. Hu, J. Ren, C. Cai, and A. Iyengar. 2020. TrajData: On Vehicle Trajectory Collection With Commodity Plug-and-Play OBU Devices. *IEEE Internet of Things Journal* 7 (2020).