

An Open-source Testbed for IoT Systems

Augusto Ciuffoletti ^a

Department of Computer Science, University of Pisa, L.go B Pontecorvo, Pisa, Italy

Keywords: Internet of Things, Docker, Energy Saving, Key-value Database, IoT Testbed.

Abstract: A research team that wants to validate a new IoT solution has to implement a testbed. It is a complex step since it must provide a realistic environment, and this may require skills that are not present in the team. This paper explores the requirements of an IoT testbed and proposes an open-source solution based on low-cost and widely available components and technologies. The testbed implements an architecture consisting of a collector managing several edge devices. Security levels and duty-cycle are tunable depending on the specific application. After analyzing the testbed requirements, the paper illustrates a template that uses WiFi for the link layer, HTTPS for structured communication, an ESP8266 board for edge units, and a RaspberryPi for the collector.

1 INTRODUCTION

When an IoT project reaches the point where empirical evidence of consistency and performance is needed, the team faces the new task of designing and implementing a testbed to carry out the experimental activity. To this end, the team needs to consider many aspects, possibly not considered at design time, vital for trial significance. Most likely, the skills internal to the workgroup do not cover some key technology.

Consider, for instance, the case of an IoT system that controls a watering system for agriculture. The task of the project consists of developing a sensor employing new technology for watering control, for instance, based on light reflection on vegetation, and the team finally develops a new device with appropriate features. After successful lab experiments, they want to deploy the system on the field with a limited number of sensors. The team, probably composed of experts in sensors and hardware design, has to design a network with a server supporting the sensors with networking, storage, control, etc. Given the limited experience in such fields, the team needs to call in new participants, or the experiment will likely exhibit weaknesses in security and data management.

This paper suggests the possibility of creating a general-purpose framework for IoT applications that enables the team in the above example to inject the new sensor in a defined and easily deployable architecture and carry out a significant experimental activ-

ity. This approach simplifies the task and improves results in several use cases as:


- application of a proposed sensor design to a given application, as seen above
- application of a proposed communication protocol in sensor communication
- application of a proposed data management in a distributed IoT system

This paper represents an early step in that direction and, for this reason, starts discussing the features of such an experimental framework. Such features correspond to the non-functional properties that need to be present to deploy a testbed. Some of them should be tunable to suit the specific application: for instance, the example above might not insist on security aspects.

After the abstract discussion, the paper describes a concrete implementation of the testbed, giving a detailed description of a template architecture: the hardware components are devices available *off the shelf*, communication uses standard protocols, the software architecture follows widely accepted conventions and tools, the code is available on GitHub.

2 RELATED WORKS

Several articles in the literature propose frameworks for the design of IoT systems. We have selected some that help to situate the content of this paper. A search

^a  <https://orcid.org/0000-0002-9734-2044>

for the *IoT framework* keywords in Google Scholar returns them as prominent, so they are likely to be representative of the state-of-the-art.

Some (Gelogo et al., 2015; Pasha and Shah, 2018) concentrate on healthcare systems, a kind of application with a remarkable social impact. It is going to gain momentum in the coming years, also because of the COVID-19 epidemic event. The term of u-health is introduced (Gelogo et al., 2015) to indicate the *ubiquitous* availability of medical assistance. The system has three components: a Body Area Network (*BAN*), made of networked sensors, an *Intelligent Medical System* that filters data from the *BAN* to identify emergency conditions, and an *Hospital System* that manages patients data and takes care of detected emergencies. Another paper on e-health IoT systems (Pasha and Shah, 2018) reveals the role of specific protocols for network management, highlighting the role played by those explicitly designed for constrained devices. Both papers stress the importance of security issues in e-health systems but fail to outline a solution. The former announces a future prototype implementation, while the second reports about the network utilization obtained from a prototype implementation using the Contiki operating system (Dunkels et al., 2004).

Security issues are the focus of other papers, also applying to a blockchain system called EdgeChain (Pan et al., 2019). Falling in the category of *permissioned* blockchain systems, it is more suitable to IoT systems, characterized by constrained devices. The IoT framework contains specific modules in charge of implementing the blockchain service (*Ethereum*) as well as a mechanism for implementing *smart* contracts that are useful to regulate access to computing and networking resources. The paper is exhaustive regarding the specific issue but disregarding the presence of constrained devices and energy-saving policies. The authors report an experimental testbed using RaspberryPi 3 Model B as edge devices and a Cisco 3850 switch.

Smart-cities are another frequently found use case for IoT systems. The provision of open-source results is regarded as relevant in a paper from the University of Bologna (Calderoni et al., 2019), which provides a reference platform that is useful as a testbed for smart-city prototypes. The platform integrates the networking and application functionalities, giving the designer full access to configuration details. The paper discusses the details of security policies introduced by AWT and Microsoft Azure, but the testbed in itself does not address the issue. Likewise, the authors do not address power-saving mechanisms.

Long Sun et al. (Sun et al., 2017) propose an ab-

stract model that makes use of microservices for the implementation of the middleware of an IoT system, managing each of them independently. The paper details the architecture of an IoT system from the sensor/actuator layer, where microservices are specialized to interact with a hardware device, up to the data management layer. The paper highlights the role of communication protocols in the interaction between microservices and edge devices. One of the services is in charge of managing security aspects. The model does not take into account energy aspects. The paper reports an experiment and illustrates the operation of a REST API that is part of it. To this end, it provides the source code of two functions: one to create a new device and another to add a trigger to an edge device.

Compared to the above articles, the present paper shares the intent of discussing the relevant features of an IoT system. In contrast, the result of such a discussion is not the generation of a collection of modules and specifications to address a specific issue but the identification of the basic non-functional features that the system must exhibit. Then we outline an abstract architecture that can be used as a reference for many use cases. The final step is a detailed description of an instance of such an abstract architecture. The implementation makes use of conventional technologies and is exhaustively defined and publicly available. One way to take advantage of such result is to inject in the testbed the non-conventional solutions under evaluation, siding, or replacing, the provided ones in a way that has much in common with the *dependency injection* used in software engineering.

3 SYSTEM MODEL

This section explores the abstract requirements of an IoT testbed without reference to a specific technology. Although experimental results strongly depend on the adopted technology, the fundamental features of the testbed, related to the specificity of IoT applications, are to some extent invariant. In this sense, adopting a shared testbed simplifies the comparison of implementations that use different technologies.

This section contains dedicated subsections for each of the non-functional requirements that needs to be present in the testbed for IoT.

3.1 Layered Architecture

A layered infrastructure is a widely adopted model to ensure the scalability of an IoT system. This model envisions edge units, sensors, and actuators as organized into clusters, each containing an infrastructure

component (here called *collector* but often indicated also as *sink* or *concentrator*). Such a component provides edge units with various kinds of services, Internet connectivity included. *Collectors* are in their turn connected to servers, either on-premises or in the cloud.

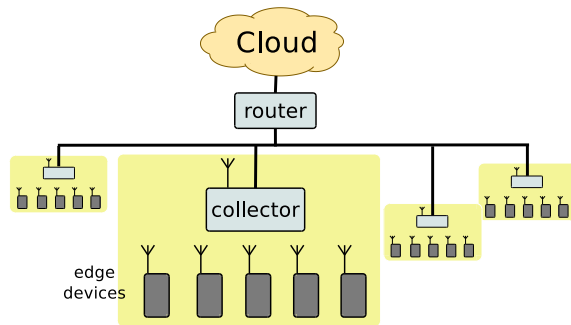


Figure 1: A typical layered architecture with adopted terminology.

The testbed illustrated in this paper covers the last two abstractions layers and proposes a template architecture with one collector serving several edge devices.

3.2 Flexible Support for Low-energy Options

There are many reasons why an IoT system must be very careful with energy consumption. When edge components receive power from batteries, that parameter determines the quantity of waste produced (in terms of exhausted batteries) and device autonomy (in terms of the life of the edge unit if the operation is unattended or of the frequency of operator intervention). When the design includes energy harvesting capabilities, the cost of generators and accumulators depends on energy consumption. Instead, when edge units receive power from the electric grid, the impact of energy consumption is considered negligible. A general-purpose testbed for IoT systems must allow the researcher to experiment with a wide range of options regarding energy.

Left alone that different boards exhibit different energy figures, duty cycling is the primary tool in the hands of the designer to reduce power consumption. Such a technique depends on the device's capability of entering a *suspend* state characterized by a substantial reduction of energy consumption. To take advantage of this, the device should remain in the *suspend* state as long as possible and wake up only when its operation is needed. The testbed should provide support to a wide range of *suspend* modes and implement

a one-fit-all solution to assist the researcher that is not specifically interested in this topic.

3.3 Flexible Support for Security

The importance of security varies from project to project. There are cases when the data gathered from edge units are sensible at different degrees, other where only credentials need to be kept secure. The testbed should provide a range of vanilla solutions, leaving the researcher free to explore other solutions within the provided framework.

3.4 Open-source

The open-source nature of the testbed is mandatory. The researcher must be able to modify any part of it and to publish solutions and experimental results without the risk of breaking copyrights. This recommendation is equally valid for software, communication protocols, and hardware design.

3.5 Modularity

Modularity is one of the ways to implement a "separation of concern" paradigm in the design of a system. In our case, this allows the researcher to conduct experiments on a specific module disregarding, as far as possible, the rest of the system. Also, the deployment of the experimental infrastructure is simplified if organized into modules with moderate dependencies.

3.6 Low Cost

Given its short-term nature, a testbed needs to be inexpensive since cost has an influence on adoption. One example addresses a researcher developing a new solution specific for one aspect of the IoT stack: a testbed that requires investments not directly related to the research focus is not attractive. Another case is that of a team that wants to reproduce a documented solution for further study. In this case, an expensive testbed might dis-encourage the researchers from exactly replicating it, which instead would be the best option from a scientific point of view.

4 A TEMPLATE FOR AN IoT TESTBED

Once defined the relevant features of a testbed for IoT projects, the next step is to implement a template.

The template is a running implementation with all the relevant details in place but with the flexibility

needed to host a specific experiment. This section outlines the design of such a system, with reference to the features introduced in the previous section.

4.1 Hardware Architecture

The reference architecture features a collector in charge of managing a set of edge devices, the sensors. The number of supported edge devices depends on the interconnection technology, but this arrangement is the basis for a scalable system, possibly replicated and rooted in a multi-layered architecture. Figure 1 illustrates a system composed of four sub-systems of this kind.

The relevant options in the implementation of such an architecture are the hardware components and the interconnection technology. The aim is that the resulting system is a *vanilla* solution that does not introduce unneeded or exotic features.

This paper adopts an ESP8266 board for edge devices and a RaspberryPi for the sink unit.

The ESP8266 is a widely available microcontroller with an embedded WiFi interface capable of running HTTPS client-side sessions. The microcontroller runs a single program stored in a Flash memory. Many boards based on such a chip simplify the flashing operation using an appropriate IDE. The Arduino open-source IDE supports the whole process, from coding in a C++ dialect to flashing.

One rationale for adopting the ESP8266 is its wide commercial availability and low cost (less than 10\$). The raw board can be readily complemented with additional capabilities using plug-in *shield* boards, but, alone, has the essential features of an edge device, as outlined below. Another with more advanced features and a comparable price is also available (ESP32), but here we opt for the one with lower power consumption.

The board embeds a WiFi interface: this greatly simplifies the hardware design, allowing minimal or no cabling. In addition, its internal design is ready for low-energy operation and provides:

- an assortment of low-energy operation modes included a *deep-sleep* with power consumption in the range of μW
- the capability of leaving low energy modes by interrupt or timer event (max 1 hour period)
- an onboard SRAM to keep persistent data during deep-sleep periods

The RaspberryPi is a one-board computer based on a quad-core ARM-Cortex processor running a fully featured operating system; the reference one is

Raspian, a Debian/Linux adapted to run on the RaspberryPi hardware. Like ESP8266 boards, it is commonly available *off the shelf* at a price ranging from 50\$ to 75\$. Depending on the model, the RaspberryPi features Ethernet and WiFi network interfaces, USB, HDMI, and Webcam plugs. The persistent memory for the operating system and the workspace are on an SD card.

When both Ethernet and WiFi interfaces are present, the RaspberryPi can be easily turned into a WiFi Access Point, possibly providing Internet access to WiFi stations. The RPi is sufficiently powerful to host database systems and other services, provided that the workload is consistent with its computing capacity.

Three RPi models are currently available: RPi 2 (2015, 900MHz/32bit), the RPi 3 (2016, 1.2GHz/64bit), and RPi 4 (2019, 1.5GHz/64bit).

The combination of ESP8266-based boards for the sensors and an RPi as a component coupling a WiFi access point and a data concentrator is a viable solution for our testbed, being open-source and inexpensive. In the rest of this section, we refine our design considering the other non-functional requirements: modularity, low-energy operation, and flexible security.

4.2 Modularity

The layered architecture is the basis for a modular (and scalable) hardware architecture. Software modularity too plays a relevant role and allows adapting the testbed to a specific task.

As a general rule, the structure of the software running on the ESP8266 board is application-specific with limited margins to introduce a generic modular pattern. The low-power modes have a well-defined and documented IDE (Espressif, 2016) so that it is pointless introducing an additional wrapper.

In contrast, the software architecture of the RPi sink is complex, and a modular organization is needed to simplify its targeting to a specific application. For instance, regarding the configuration of the network layer protocol (e.g., replacing WiFi with LoRaWAN), the application layer protocol (e.g., replacing HTTP with CoAP), the storage management (e.g., using a relational DB or other ways to manage sensors data).

A dockerized architecture suits the need for modularity and easy configuration. The designer requiring a given operation plugs in the appropriate dockers and interconnects them in a virtual network. Following such an approach, the RPi runs the Docker hosting software, and the desired Dockers are loaded and run. Dockers can be either available from the hub

(<https://hub.docker.com>) or designed on purpose and easily shareable on multiple installations (consider, for instance, a testbed with multiple concentrators). The configuration instructions, networking included, are written in one single *docker-compose.yml* file.

The adoption of the Docker technology is limited to configuration options that do not impact hardware devices. For instance, switching from WiFi to Lo-RaWAN cannot be easily managed with the Docker technology. In this case, it is probably better to leave the task to configuration scripts for the specific technology.

4.3 Flexible Low-energy Configuration

There are two kinds of low-energy operation: soft and deep sleep. During a soft-sleep period, the fundamental computing capabilities of the system are operational or in a low-power state, while energy-consuming ones are disabled. For instance, the ESP8266 has two sleep modes that can be considered as *soft-sleep* (Espressif, 2016):

- modem sleep: the WiFi modem is off when unused, keeping into account the need to receive beacon frames (the frequency of which is in the range of seconds, configurable in the Access Point)
- light sleep: suspends the CPU and disables the modem

The configuration of a specific soft-sleep policy has a limited impact on components outside the sensor itself. For instance, to tune the beacon frequency to extend modem-sleep periods.

In contrast, the management of deep-sleep periods has an impact on system-level design. The reason is that, during a deep sleep interval, most of the chip components do not receive power at all. A relevant effect is that the working memory loses its content. Another is that the system clock stops measuring time. To avoid zero-knowledge wake-up, the sensor needs to backup working data and record their address to retrieve them when waking up. Similarly, a networked component provides the *time of the day* and the calendar date (if needed by the application).

Most modern microcontrollers (the ESP8266 included) embed the functionalities needed to implement an efficient deep-sleep policy. Such support consists of a clock with a small RAM kept powered even during deep-sleep periods. As a general rule, their characteristics are not for general-purpose solutions, but they provide substantial help: for instance, in the case of an ESP8266, the clock accuracy is in the percent range, and persistent memory capacity is

of a few KBytes. Therefore the clock is appropriate to time a periodic wake-up, and the memory can save a link to external storage.

In the proposed architecture, the component providing storage and time services is primarily the concentrator. A Linux server can provide various kinds of clock synchronization services, depending on the accuracy required by the application. The Network Time Protocol (Mills et al., 2010) is easily installed and may provide microsecond synchronization on suitable networks. If sensor operation only requires a generic time of day with a one-second precision, a viable alternative is piggybacking timestamps to other messages.

Regarding the support for the remote storage of edge devices, a suitable solution consists of using a REDIS <https://redis.io/> key-value storage: the key, a random string, is recorded in the persistent RAM on the edge component, while the value can be any sort of — possibly bulky — data, like an image, a sound spectrum, or a GPS track, that is associated to the key in the REDIS database.

According to the Dockerized structure of the concentrator, a Docker container with one of the REDIS images found in the hub implements the database.

The REDIS database access is through a REST server hosted on the concentrator. The HTTP service should offer three basic CRUD operations associated with REST verbs:

- create a new key (PUT)
- record a value associated with a key (POST)
- download the value associated with a key (GET)
- deleted an unused key and the associated value (DELETE)

The design of the REST server follows any server-side web framework: the template uses Flask/Python. A custom Docker image embeds the server together with the Python libraries and other dependencies. In this way, the deployment of architectures containing several sinks is straightforward.

It is appropriate to introduce an HTTP proxy in front of the Flask server to improve server security and performance. The popular Nginx proxy is well suited for this task and is available as a Docker image.

At this point, the internal structure of the template testbed, summarized in figure 2, is sufficiently defined.

4.4 Flexible Security Support

The relevance of security aspects changes depending on the application. A watering plant produces information that is of limited interest for a potential in-

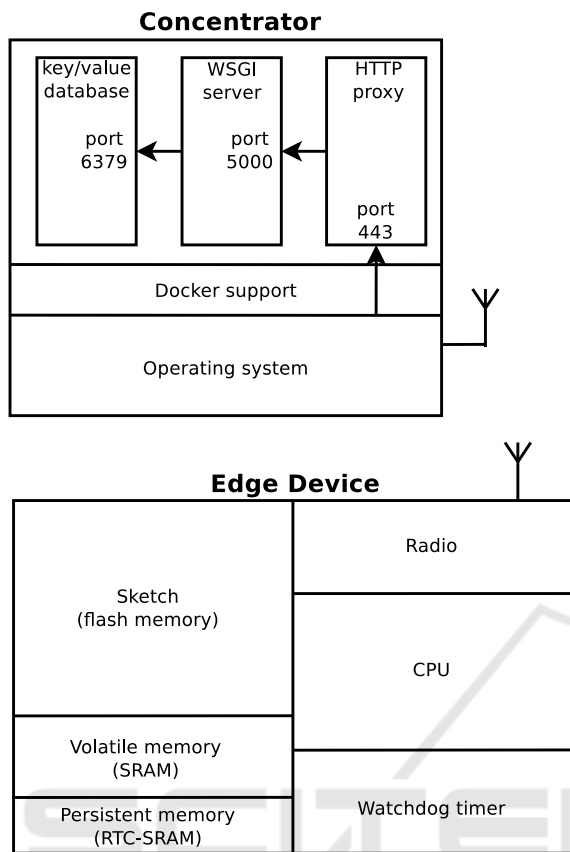


Figure 2: Outline of the internal structure of the Concentrator and of the Edge device.

truder, but its design should nonetheless avoid denial of service attacks. In contrast, an automotive control has to be more cautious about the possibility of an intrusion. In the extreme, a medical system has to protect every piece of information it produces, especially when it can be associated with an individual.

For such reason, the rules to access the key-value service depend on the application, so that the template has to be flexible enough to implement a range of solutions.

A key-value system natively supports mild security. If the number of keys is sufficiently high and their values are well randomized, then it is *difficult* for an intruder to find a used key and take the place of a trusted sensor. This argument allows claiming the GET, POST, and DELETE verbs are sufficiently secured, but not the PUT, which creates a new key/value pair. The attack pattern, in this case, consists of gaining access to the network and issue PUT requests to introduce noxious data or saturate database capacity. In conclusion, securing the PUT operation is relevant even for mildly critical applications. The mediation of a trusted component, which authorizes a PUT re-

quest, solves the problem. Such a component may be an operator equipped with a smartphone with an authentication mechanism or a computer that flashes an assigned key in the mote Flash memory.

The presence of a trusted component in the chain does not guarantee against the possibility that this same component steals a valid key, thus acquiring the capability to configure an intrusive edge device or to access data that can be associated with a specific sensor. For instance, this may generate a leakage of sensitive data in an e-health system. A way to avoid this problem consists of changing the key at each GET operation to make meaningless the — possibly stolen — initial key. According to this mechanism, upon receiving a legal GET request, the server changes the key associated with the value and returns the requester the value together with the new key. This technique also marginally improves the security level provided by a random key of a given length for the following reason. Without the *key-swapping* mechanism, the intruder certainly finds a valid key by scanning all possible keys. Instead, with the *key-swapping* mechanism in place, the scanning technique is not successful since any key may become valid after having been discovered as non-valid.

The above mechanism does not protect against the case that data stored in the database are stolen by physically removing the SD card from the concentrator device. Although the data cannot be associated with a defined sensor, the anonymized data can be equally considered sensible for the organization. In this case, the sensor should encrypt the data associated using a key, known solely to the sensor itself, generated when the mote is switched on, recorded in the persistent RAM, and used at each POST (to encrypt), and GET (to decrypt) operation.

With the above mechanisms in place and using the HTTPS protocol, the testbed is resistant to a wide range of attacks. But, in case of failure of an edge device, there is no way to recover orphaned data. Finding a compromise between fault tolerance and security is a task that a flexible IoT testbed helps to solve.

4.5 Experimental Results and Code Availability

The tests run on a prototype confirm that the testbed is suitable for generic experimental setups. One relevant aspect is the capacity of the Raspberry Pi to support a significant number of hosts. We have evidence that the device can support a load of 100 edge devices each running an operation every 10s with a limited impact on service times. In contrast, the Raspberry Pi has limits when operating as an access point, and inde-

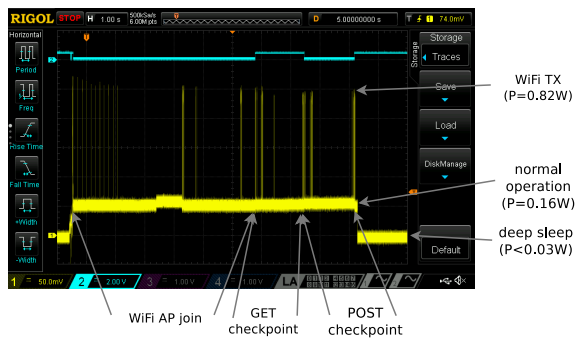


Figure 3: Trace of one operation cycle on the Wemos board - Current consumption is measured on a 1 Ohm shunt in series with power supply (3.3V), which introduces an observable noise.

pendent tests¹ claim that it cannot support more than 19 simultaneous connections. A typical operation cycle with a negligible time spent in payload operation has a duration of approximately 8 seconds, distributed as follows: 5 seconds to join the WiFi infrastructure, and 1.5 seconds each for the GET operation to retrieve a checkpoint from the Redis Database, and the POST operation to store a new checkpoint back into the database.

The software for testbed edge and collector devices is available from two public Github repositories: one is dedicated to the ESP8266 (https://github.com/AugustoCiuffoletti/IoTTemplate_esp8266), another to the software running on the Raspberry Pi (<https://github.com/AugustoCiuffoletti/kviot>).

5 CONCLUSIONS

IoT systems involve a stack of technologies, from the physical layer to the application, with specific requirements: a research effort usually targets one of such aspects. To validate a result, the research team has to produce a testbed to run experiments, frequently overlooking the details that are not the research focus: this may damage the validity and interest of experimental results. This paper deals with the design and implementation of a testbed for IoT systems that allows the designer to focus on one aspect injecting the new piece of technology in a framework designed *by convention* that takes into account scalability, energy-saving, and security.

We have currently tested the soundness of the solution described in this paper. A Research-

¹There are many posts on this topic with unchecked experimental data. A reliable one might be <https://raspberrypi.stackexchange.com/questions/50162/maximum-wi-fi-clients-on-pi-3-hotspot>

Gate project with links to GitHub repositories will report about developments: "A framework for IoT systems" (<https://www.researchgate.net/project/A-testbed-for-IoT-solutions>).

REFERENCES

- Calderoni, L., Magnani, A., and Maio, D. (2019). Iot manager: An open-source iot framework for smart cities. *Journal of Systems Architecture*, 98:413–423.
- Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki: a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*, pages 455–462. IEEE.
- Espressif (2016). *ESP8266 Low Power Solutions*. Espressif Systems IOT Team, v 1.1 edition.
- Gelogo, Y. E., Hwang, H. J., and Kim, H.-K. (2015). Internet of things (iot) framework for u-healthcare system. *International Journal of Smart Home*, 9(11):323–330.
- Mills, D., Martin, J., Burbank, J., and Kasch, W. (2010). Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard).
- Pan, J., Wang, J., Hester, A., Alqerm, I., Liu, Y., and Zhao, Y. (2019). Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts. *IEEE Internet of Things Journal*, 6(3):4719–4732.
- Pasha, M. and Shah, S. M. W. (2018). Framework for e-health systems in iot-based environments. *Wireless Communications and Mobile Computing*, 2018.
- Sun, L., Li, Y., and Memon, R. A. (2017). An open iot framework based on microservices architecture. *China Communications*, 14(2):154–162.