Simulation and Benchmarking of IoT Device Usage Scenarios Using Zephyr and Qemu

Bill Schirrmeister, Frank Geyer, Steffen Späthe

Friedrich Schiller University Jena
Department of Computer Science, Software Engineering Group
Jena, Germany
Email: bill.schirrmeister@uni-jena.de
frank.geyer@uni-jena.de
steffen.spaethe@uni-jena.de

***Abstract*—The development of a device command and control infrastructure for Internet of Things devices with a focus on low resource consumption is the primary goal of the research project "unic²ast". This paper proposes a simulation environment that enables to carry out benchmarks of this infrastructure according to a variety of application scenarios. The presented simulation setup uses Qemu virtualized embedded devices with an application based on Zephyr OS. The basic suitability of the approach is demonstrated and potentials for further development are identified.**

Keywords–IoT Device Management; Lightweight M2M; Qemu; Zephyr; Device Simulation.

I. INTRODUCTION

The number of connected devices in the Internet of Things (IoT) is forecast to reach over 75 billion by 2025 [1]. As the overall goal of the research project unic²ast [2] a device command and control infrastructure (DCCI) for scenarios with a large number of IoT devices is meant to be realized.

Following the well-known client-server-model, a server infrastructure for managing and administrating IoT devices which provides transparency with regard to their concrete application scenarios is being developed. A good system performance is considered to be essential for this project, since it is required to manage and integrate a potentially large amount of devices (a) with possibly constraint hardware resources (b). To evaluate this system property, development of a simulation system is one of the project goals. This should allow for the measuring of values that are as realistic as possible and help to identify bottlenecks in the processing inside the DCCI system.

Available network simulation systems such as "ns-3" [3] or "OMNeT++" [4] are not sufficient for unic²ast, since models of the server and the processing logic would need to be created. This would lead to a rather difficult realization and cannot be achieved without knowledge about the runtime behavior of the individual components. The term "individual components" also includes the operating system, hardware (input / output to RAM, hard disk, etc.), runtime environment (e.g., Java VM), database management systems (MariaDB, Postgres, etc.), and other web services. Likewise, unic²ast has to use specific protocols (such as OMA Lightweight M2M (LwM2M)) in order to be able to provide the desired system functionalities. Existing network simulation systems do not offer out-of-the-box implementation of such specific IoT device protocols.

Unic²ast relies on the use of a real DCCI instance that is occupied by virtual IoT devices (i.e., simulated load) in order to test the remote maintenance of multiple IoT devices via the DCCI with a focus on performance and stability analysis. The simulation is coordinated by a test coordinator, i.e., a software solution for planning, controlling and evaluating the simulation runs. Such a system structure is not uncommon in practice. Standard load test systems for server applications such as Apache JMeter [5] or Gatling [6] are built according to the same principle.

Unic²ast follows an approach where virtual IoT devices are used to test the real DCCI server implementation. In principle, one could roughly calculate load limits or determine them by using purely virtual simulations. However, it was opted for the possibly more resource-intensive evaluation approach because the server system must be able to handle a large number of any IoT devices in productive operation. Therefore, it is not possible to predict the need for communication in the form of static information, as required by other simulation approaches. In particular, bottlenecks in internal communication processes inside the device command and control infrastructure are unknown in advance of the load tests.

This paper is not intended to provide a comprehensive test specification for the DCCI. Likewise, there is no complete specification of the system's properties to be tested and no detailed definition of test procedures.

The aim of this paper is to present the infrastructural approach for the parallel execution of several, configurable client instances for a given application scenario with focus on benchmarks of the overall system. On the basis of the explanations here, it should be possible at further steps to define concrete and more complex test sequences and to carry out corresponding data collection with focus on performance-benchmarks at runtime.

We have organized the rest of this paper in the following four sections. The core requirements to the benchmark infrastructure and available implementation approaches are presented in Section II. Section III is about how the selected infrastructure approach was implemented in detail. Aspects of

 ISBN: 978-1-61208-805-1

a practical usage are shown in Section IV. At the conclusion of the paper, in SectionV we summarize the results achieved and briefly evaluate the chosen approach based on our formulated requirements.

II. Requirements and Implementation Approaches

To prepare the selection of a suitable approach and environment for simulation within unic^{2}ast, the following essential requirements to the benchmark infrastructure were defined:

- simulated devices behave quite similar to original devices
- ability to run multiple simulated device instances on the same host
- LwM2M connection to the system under test
- parameterization of the device instances
- retrieval of runtime information after startup
- automation of the device instance lifetime and the test procedure

OMA Lightweight M2M (LwM2M) in this context is a protocol of Open Mobile Alliance (OMA) for IoT device management [7]. The LwM2M protocol defines an application layer communication protocol between client and server and focuses on low resource consumption. LwM2M is based on Constrained Application Protocol (CoAP).

As described before, the benchmark infrastructure should use simulated client devices. A valid approach would be to implement the client's functionality within a dedicated simulation program. The program could be implemented in a comfortable high-level language, and all available libraries could be used. Furthermore, this program could use multithreading to simulate several individual devices simultaneously. This simulation program could, therefore, be implemented with relatively little effort. This approach would also lead to a rather simple test setup.

On the other hand, an application program already exists for the target hardware, which realizes the communication between server and embedded system. This embedded program itself is its best simulation in the context of a load test of the overall system. However, the use of several hundred real devices is inappropriate. Therefore, another approach to build a test system setup is to emulate the embedded devices, including their existing application logic. In this way, the additional parallel development of a synthetic load driver based on an additional development environment can be avoided.

III. Approach

A combination of shell scripts with "Zephyr OS" (Zephyr) and "Quick Emulator" (Qemu) was used to implement a solution that takes the given requirements into consideration. Zephyr OS is an open source real-time operating system for IoT devices with a small memory size and fixed hardware configuration [8]. Qemu is a free virtualization software for complete hardware emulation [9].

Figure 1 shows the resulting system landscape. The system to be tested is shown inside the *System under test* box as *DCCI*. However, the actual structure of the system to test is considered to be a black box and therefore was omitted. The landscape to be tested can consist of complex server structures.

The virtual system part (box *Simulated devices*) usually consists of several IoT devices that are implemented as Qemu processes executed in parallel. Each Qemu process simulates exactly one IoT device that is operated by a Zephyr OS based embedded application. For communication purposes between virtual devices and the system under test via LwM2M, an extended version of the LwM2M client included in Zephyr is used. The underlying Internet Protocol (IP) connection of clients is realized by setting up a virtual bridge connecting them with the hosts physical ethernet adapter.

Every virtual device is assigned one named pipe for input and output which is set up in the hosts file system. Their coupling with the respective Qemu process is realized by configuring it as a serial port when calling the Qemu executable. On the Zephyr OS side, this port is connected to the built-in shell subsystem. This makes it possible to not only send shell commands to each device, but also to record the shell output.

Functions for automating preparation, execution and post-processing of benchmarks are implemented in Bash scripts. A higher-level coordinator script defines the test structure and sequence and therefore uses the functions modularized in other scripts. The higher-level script thus represents the implementation of a concrete test scenario. Other scenarios can be realized by alternative implementations of this script.

In the following part the most important functions, based on prior defined requirements, will be discussed in more detail.

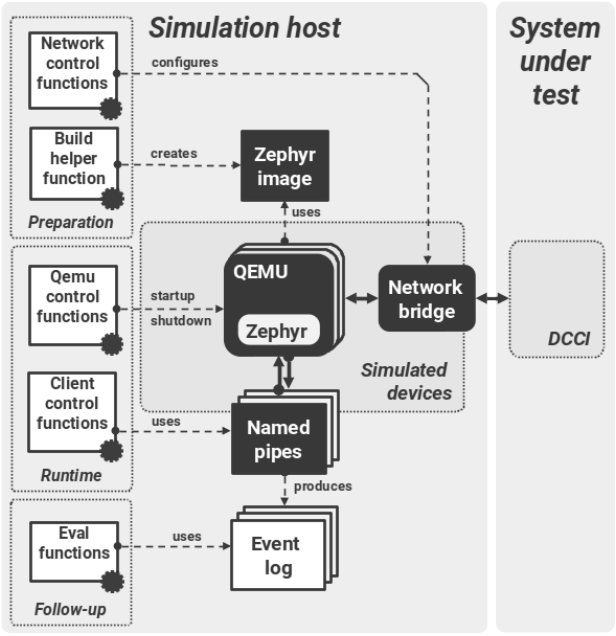

Figure 1. Simulation-landscape at unic^{2}ast

A. Parallel Execution on a Single Host

Parallel execution of multiple test clients on a single host is done by launching several virtual machines at once.

The unic^{2}ast project makes use of the `qemu_x86` board configuration shipped with Zephyr source code. Following the Zephyr documentation, the application program that implements the LwM2M client is developed based on the Zephyr kernel. It gets compiled into a single binary file alongside the

kernel to be then executed in a virtual machine. Execution is thus isolated from the host and other virtual machine instances. The compilation is done by calling `cmake` with the parameter

```
-DBOARD=qemu_x86
```

followed by an additional call to `make`. After successful compilation, the resulting binary file is then used when launching a Qemu process. This may be done by calling the respective Qemu executable via

```
qemu-system-i386 -kernel [PATH-TO-IMAGE]
```

providing it with the built Zephyr based application image via parameter `kernel` (additional parameters omitted due to layout restrictions).

By calling Qemu multiple times successively, several Qemu processes can be lunched in parallel, each executing a client in its own virtual machine. To avoid the need to execute each call to Qemu individually on the command line, this was automated within a parameterizable loop as part of a Bash function.

A call to the specified function results in the following behavior

- creation of a named pipe per client
- generation of a unique medium access control (MAC) address per client
- startup of the specified number of Qemu processes, utilizing the built Zephyr based application image as described above, and assigning the named pipe to the serial port and the MAC address to the ethernet adapter respectively
- calling of `cat` on the output pipe for each client and redirecting its output stream to a file
- storage of the meta data describing the clients and other relevant data during runtime in corresponding files

Similarly, a shutdown function takes care of the scheduled shutdown of the started instances using the process ID noted as part of the meta data for each Qemu process.

B. Connection to Target System

The client's ability to connect to the target system using the IP is a necessary prerequisite for the required LwM2M connection, which is based upon CoAP and is achieved through the following four steps:

- creating a network bridge on the host
- providing an ethernet adapter to the Zephyr image
- linking the Qemu machine's ethernet adapters to the bridge on startup
- allocating a suitable unique IP address to each client

The possibility of setting up a network bridge as a virtual connection element on layer 2 of the Open Systems Interconnection (OSI) model is already built into modern Linux operating systems. The necessary bridge can therefore be set up in a terminal via

```
ip link add [BRIDGE] type bridge
```

After assigning an IP address to the host to ensure its reachability on the bridge, the host's physical ethernet adapter is also connected to the bridge via

```
ip link set [ADAPTER] master [BRIDGE]
```

The bridge thus forms a virtual extension of the physical network on the MAC layer that is executed on the host.

To successfully connect the clients to the bridge, they need an ethernet adapter too. According to Zephyr OS documentation for building `qemu_x86`, the source code required for the virtual ethernet adapter `E1000` can be included in the compiled image by specifying the overlay file `overlay-e1000.conf` alongside the others with the parameter `-DCONF_FILE` when `cmake` is called.

The Qemu machines are then connected to the bridge by specifying

```
-nic bridge,br=[BRIDGE],mac=[MAC-ADDR]
```

during the call to the Qemu executable, where `MAC-ADDR` is the generated unique MAC address of a client.

In addition to the connection already described on the MAC layer, the clients still need a unique Internet Protocol version 4 (IPv4) address in order to be able to establish CoAP based connections to the target system. A static assignment as in the LwM2M sample found in Zephyr's sources is unfavorable because the address would need to be configured at compile time. If then arbitrary `n` instances of the client were desired to be assigned an address, potentially expensive `n` compilation runs would be necessary. The dynamic assignment of addresses via Dynamic Host Configuration Protocol (DHCP) has also proven to be impractical. With both a local DHCP server on the host and an external server on the network, it was impractical to reliably serve IPv4 addresses to a large number of volatile clients (in the hundreds).

Our solution to this problem uses static address assignment by means of a Zephyr shell command combined with address generation using `prips`. The command line program `prips` prints a line-by-line listing of all IPv4 addresses of a given IPv4 subnet. This subnet has to be specified in Classless Inter-Domain Routing (CIDR) notation when calling `prips` as follows

```
prips "192.168.0.0/24"
```

Substracting at least the subnet's first address (network) and the last one (broadcast) is necessary afterwards, since they must not be used for client assignment. Further addresses may have to be omitted if they are already assigned to the client's host or other computers on the physical network, resulting in the final list of usable addresses. The assignment to the clients is then carried out by a dedicated Zephyr shell command, which was implemented specifically for that purpose. It then assigns the address inside its callback by calling the function

```
net_if_ipv4_addr_add_by_index
```

which is provided by Zephyr's network stack accordingly.

C. Parameterization and Value Retrieval at Runtime

A named pipe each for input and output streams in the host's file system are used for direct communication between

the host and the virtual machines during runtime. Splitting into two file paths is necessary because buffer conflicts could occur if only a single path is used for both input and output. The separate pipes have the same base file name, but differ in their extension, being assigned an `.in` or `.out`. The assignment of such a "double pipe" to its dedicated virtual machine then is made by specifying just the base file name as parameter value for the serial port when calling the Qemu executable. The host operating system automatically separates the input and output character streams to the separate files accordingly.

The parameterization of a client at runtime is then carried out by the transmission of specific shell commands to each Qemu machine. This may be achieved by using `echo` like

```
$ echo "[COMMAND]" > [PATH]/[PIPE].in
```

redirecting it to the respective pipe's input path. In return, the pipe's output path enables all shell outputs of a client to be recorded, for example by creating a `cat` process of the form

```
$ cat [PATH]/[PIPE].out >> [PATH]/[FILE] &
```

Since the basic shell commands available in Zephyr OS are insufficient for the purposes described here, application-specific commands had to be implemented.

The shell interface of a Zephyr application follows a hierarchical structure. Several subordinate command words can be assigned to a higher-level command word to form a specific command. Parameter values that have to be transmitted alongside a command can be specified on the leaves of the defined tree. The callback method, which is also to be specified on the leaves, is called when the respective command is received, providing it with the transferred parameter values and thus allows the behavior of the command to be implemented within the Zephyr application. Among other things, registering a command on the Zephyr shell subsystem can be done through special C macros [10].

The concrete commands for benchmarking implemented in this concept allow the necessary configuration of parameters that are required for the secure connection to the target system (e.g., Datagram Transport Layer Security (DTLS) pre-shared key). In addition to that, virtual temperature sensors may also be created on each client, the values of which are randomly varied at random time intervals in order to generate a realistic load by means of subscriptions made by the server.

Input or output via named pipes with the above shell commands can be issued manually on the host's shell. However, these processes were largely automated by script functions.

D. Client Identities and Resource Subscriptions

For securing the client's CoAP/LwM2M connection to the target system by means of the DTLS protocol within the context of the uni^{2}ast project, the pre-shared key approach was to be used.

This key which has to be unique for each client must be known to the server in advance alongside its unique identifier string and the LwM2M endpoint name. In order to carry out load tests of the target system with varying multitudes of these clients information stored to the target's database. Therefore, it is necessary to provide functions that enable this data

- to be reported to the target system at the start of each individual test run and
- to be deleted from the target system after their end, respectively.

For the same reason and because of the number of clients in a test run being potentially large, it is also required to automatically generate the client's identity and security information parts, as mentioned above.

Randomly generating these values is undertaken in the implementing Bash function by read access to `/dev/urandom` of the Linux host. They are collected in a Comma Separated Value (CSV) file and are thus available to the other implemented Bash functions that need to have access to them, for example the Bash functions explained in Subsection III-C that are used for parameterizing the clients. Furthermore, the functions responsible for reporting the clients information to the target and deleting them later are implemented to use a Representational State Transfer Application Programming Interface (REST API) provided by the target specifically for that purpose.

IV. Practical Usage

To demonstrate the presented solution, a simple test scenario defined for the real DCCI system implementation developed in unic^{2}ast and the experiences made when preparing and performing the benchmark are given below.

A. Example Test Scenario Definition

The objective of the chosen scenario was to allow for the measurement of the target system's response times as seen by the clients, as well as the resource requirements of the target system in the context of an increasing number of clients from test run to test run. Based on the assumption that the regular registration update sequences between each client device and the DCCI server and the notification messages sent by the clients due to server side subscriptions for changing values are sufficient to cause a certain load on the target system, the scenario was defined to provide that

- each client simulates two temperature sensors (instances of the Internet Protocol for Smart Objects (IPSO) object 3303) with values adjusted at random intervals and
- the server subscribes to changes of these values as soon as the clients are connected.

Data on the target system's performance should then be gathered by measuring the registration event response latency of the server on client side and the server's computing and working memory utilization on server side during a fixed time interval the coordinator script has to wait during each run of the test. The procedure in this scenario should consist of four consecutive runs, starting with 100 clients during the first run, increasing to 400 and 800 throughout the second and third runs and ending up with 1,000 clients during the last run. The waiting period before shutting down the clients at the end of each run was set to 90 seconds.

 ISBN: 978-1-61208-805-1

B. Coordinator Implementation

The scenario described was implemented as a coordinator script, according to the principle explained in Section III, and uses the Bash functions implemented as part of the solution. The practical process flow for test setups is shown in Figure 2.

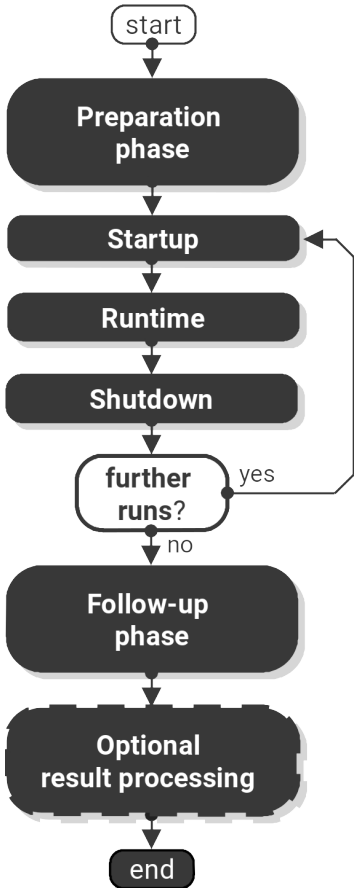

Figure 2. process flow in unic^{2}ast test setup

The Preparation phase includes tasks that are necessary for the test execution, but are not to be repeated as part of the test in each run. This applies, for example, to build the Zephyr-based application image for the virtualized clients, creating the network bridge on the host but also booting up the target system and any other necessary steps to ensure that the target system can be reached from the test host.

Similarly, the Follow-Up phase includes all activities that are carried out independently of the individual test runs after the completed test. This includes for example removing the network bridge, possibly shutting down the target system, and evaluating the log files generated on the target system and on the clients by using R.

The three main phases, however, depend on the design of the specific test scenario.

The Startup phase includes at least:

- generating the security information of the clients and announce them to the target system
- starting the clients and assigning the respective IP addresses
- configuring the clients with the respective security information and the simulated temperature sensors

The following Runtime phase includes at least:

- connecting the clients to the server via LwM2M registration
- setting up the subscriptions by the server
- waiting for the period of time specified before the start of the test
- disconnecting all clients via LwM2M deregistration

The end of a test run with the Shutdown phase is indicated by:

- terminating the Qemu processes
- deleting the client's security information from the server
- collecting the log files of all clients and the server

The implemented coordinator script realizes the main phases that are repeated during each run inside a loop and further collects the files containing the measurement data gathered on client side and on server side. The data evaluation based on that files after the test's completion was conducted by specifically implemented R scripts.

C. Results and Limitations

The subsequent evaluation of the collected data within the unic^{2}ast project enabled conclusions about the suitability of the DCCI system regarding its specific requirements, but are not subject of this paper. However, by benchmarking the real DCCI system according to the scenario defined above, it was possible to also draw conclusions about the suitability of the presented benchmark solution itself, which are explained below.

The test's preparation and execution revealed some practical limitations of the presented approach, particularly with regard to the implementation in Bash and the usage of Qemu virtualization.

Firstly, the virtual clients cause a relatively high overhead in resource consumption. Regarding the amount of working memory, prior experience with lower numbers of clients already showed that each virtual client must be started with at least 16 MB RAM for the application to be stable, but in this case approximately 25 MB of host RAM are actually used per client, which speaks for a relatively high overhead due to the virtualization using Qemu. In order to carry out tests with the scenario described above with up to 1,000 clients, we therefore decided to use a host machine configuration equipped with 32 GB working memory, also taking into account some reserves to be used by the host's operating system and still leaving about 5 to 6 GB of RAM free. Although the maximum number of clients was limited by the scenario, we aimed to further increase their number using the remaining free memory. But regarding the CPU usage, our tests showed this to be a limiting factor as well. Using a machine configuration with 8 CPU cores we could not run more than about 1,020 clients. We suspect that this is due to peaks in CPU usage during specific moments in the client lifecycle, e.g., during Zephyr kernel startup, configuration parameter provisioning, and temperature sensor simulation. However, the chosen system configuration proved to be sufficient for load tests with up to 1,000 virtualized clients. Ultimately, despite these observations, the

 ISBN: 978-1-61208-805-1

essential positive aspect of this approach should not be underestimated. The development of the test application brought valuable insights into the Zephyr operating system and its numerous interfaces. It can be considered close to the real software development for embedded solutions. This can also be particularly advantageous in the further course of the unic^{2}ast project.

Secondly, while Bash scripting proved to be a suitable way to quickly adapt specific ideas into reality, it also clearly showed limitations. When the amount of source code grows over time during a software project, it is likely to run into specific issues regardless of the programming language used. Though simple modularization of Bash source is possible by separating them over several files, it lacks more sophisticated means to manage evolving dependencies. Furthermore, the absence of multi-dimensional arrays and the passing of data between different functions, both being essential elements in software programming, turned out to be challenges when using Bash. Ultimately, however, the solution could still be implemented, which is also be due to the many helpful tools of the Linux ecosystem, e.g., `prips`.

V. Conclusion

The test setup described in the section above was successful overall and the listed requirements (see Section II) are demonstrably met by the presented approach:

- Qemu virtualization is able to run Zephyr OS based application.
- Qemu virtualization is used to run multiple Zephyr devices in parallel on the same host.
- The LwM2M client in Zephyr serves as the basis for the development of a client application that enables a DTLS secured connection to the target system and can generate additional load on the target system through simulated LwM2M resources.
- By using named pipes on the host, a bidirectional connection with the Zephyr OS based applications shell subsystem is used for the runtime parameterization of the clients and the interception of all outputs.
- The solutions presented are automated in the form of Bash script functions and can therefore be used in different test scenarios.

However, the test also revealed some limitations of the approach in terms of implementation difficulties due to Bash scripting and high resource consumption due to the overhead introduced by the virtualization mechanism.

Considering a possible further development of the presented benchmark setup, we suppose a replacement of the Bash based coordinator to be reasonable.

This would not only allow to use a advanced programming language like Java and its potentially large ecosystem of tools and frameworks. Moreover, it could let us implement the solution as an adapter for integration with existing tools specialized in load testing tool like JMeter. This allows us to leverage their potentials while not loosing our ability to create realistic test scenarios based on real-world embedded software components. Over more scattering virtual client devices among multiple host systems could be realized. That will decrease the resource-based limits in the number of simulated devices.

Despite the high overhead in resource consumption, the use of emulated embedded devices with real application software has proven to be practical. The additional development of synthetic devices in network simulators or other specialized frameworks is not necessary. At the same time, the realized device behavior is quite close to the real world without the additional maintenance of a second code base. Furthermore, our approach tests the actual DCCI implementation and not a statistical simulation of it. Therefore, we expect test results which correspond to the real application.

On an overall view, our approach fulfils our formulated requirements and enables us to implement more complex load tests with minimum effort and the greatest possible accuracy.

References

[1] Statista Research Department. Internet of things - number of connected devices worldwide 2015-2025. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ [retrieved: Dec., 2019]

[2] unic^{2}ast Projekt. [Online]. Available: http://swt.informatik.uni-jena.de/Projekte/ [retrieved: Aug., 2020]

[3] nsnam. ns-3 — a discrete-event network simulator for internet systems. [Online]. Available: https://www.nsnam.org/ [retrieved: Feb., 2020]

[4] OpenSim Ltd. OMNeT++ discrete event simulator. [Online]. Available: https://omnetpp.org/ [retrieved: Jan., 2020]

[5] Apache Software Foundation. Apache JMeter. [Online]. Available: http://jmeter.apache.org/ [retrieved: Dec., 2019]

[6] Gatling Corp. Gatling Open-Source Load Testing - For DevOps and CI/CD. [Online]. Available: https://gatling.io/ [retrieved: Jan., 2020]

[7] Open Mobile Alliance. Lightweight M2M (LWM2M) - OMA SpecWorks. [Online]. Available: https://www.omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m/ [retrieved: Aug., 2019]

[8] Zephyr Project. [Online]. Available: https://www.zephyrproject.org/ [retrieved: Nov., 2019]

[9] F. Bellard. QEMU. [Online]. Available: https://www.qemu.org/ [retrieved: Jan., 2020]

[10] Zephyr Project. Shell. [Online]. Available: https://docs.zephyrproject.org/2.1.0/reference/shell/index.html [retrieved: Apr., 2020]

 ISBN: 978-1-61208-805-1