Contention Aware Web of Things Emulation Testbed

Raoufeh Hashemian
rhashem@ucalgary.ca
University of Calgary, Canada

Niklas Carlsson
niklas.carlsson@liu.se
Linköping University, Sweden

Diwakar Krishnamurthy
dkrishna@ucalgary.ca
University of Calgary, Canada

Martin Arlitt
martin.arlitt@ucalgary.ca
University of Calgary, Canada

ABSTRACT

Since the advent of the Web, new Web benchmarking tools have frequently been introduced to keep up with evolving workloads and environments. The introduction of Web of Things (WoT) marks the beginning of another important paradigm that requires new benchmarking tools and testbeds. Such a WoT benchmarking testbed can enable the comparison of different WoT application configurations and workload scenarios under assumptions regarding WoT application resource demands and WoT device network characteristics. The powerful computational capabilities of modern commodity multicore servers along with the limited resource consumption footprints of WoT devices suggest the feasibility of a benchmarking testbed that can emulate the application behaviour of a large number of WoT devices on just a single multicore server. However, to obtain test results that reflect the true performance of the system being emulated, care must be exercised to detect and consider the impact of testbed bottlenecks on performance results. For example, if too many WoT devices are emulated then performance metrics obtained from a test run, e.g., WoT device response times, would only reflect contention among emulated devices for shared multicore server resources instead of providing a true indication of the performance of the WoT system being emulated. We develop a testbed that helps a user emulate a system consisting of multiple WoT devices on a single multicore server by exploiting Docker containers. Furthermore, we devise a novel mechanism for the user to check whether shared resource contention in the testbed has impacted the integrity of test results. Our solution allows for careful scaling of experiments and enables resource efficient evaluation of a wide range of WoT systems, architectures, application characteristics, workload scenarios, and network conditions.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; • **Computer systems organization** → *Client-server architectures.*

1 INTRODUCTION

In recent years, the Internet of Things (IoT) has evolved substantially. One particular change has been an integration with Web technologies. This evolution, dubbed "The Web of Things" or WoT for short, motivates new benchmarking tools and frameworks to facilitate performance evaluation of WoT systems. Since the advent of the Web, new Web benchmarking tools have frequently been introduced to keep up with evolving workloads and environments. For example, new benchmark suites have been developed to address changes such as Web 2.0 [57], Semantic Web [41], multicore Web infrastructure [29] and cloud computing [47]. The introduction of WoT marks the beginning of another important paradigm that requires new benchmarking tools.

While real testbeds are most suitable for testing the performance of already implemented applications on a platform close to its intended hardware, testing on real testbeds is not always feasible, e.g., due to limited time frame and budget. Therefore, simulation and emulation approaches are typically required when a system is in the design and implementation stage. However, existing solutions [22, 38, 51] do not yet provide an integrated mechanism to evaluate how the performance of large WoT systems can change as a function of the overall system architecture, application characteristics, workload, device characteristics, and network conditions. This gap motivates the need for new approaches.

The powerful computational capabilities of modern commodity multicore servers along with the limited resource consumption footprints of WoT devices suggest the feasibility of a benchmarking testbed that can emulate the application behaviour of a large number of WoT devices on just a single multicore server. However, benchmarking testbeds can be latency sensitive. Consequently, the testbed needs to detect and handle the implications arising out of contention for shared resources in the multicore server, as such contention can have a significant adverse effect on the validity of the performance results reported by the testbed.

The first challenge in designing a WoT benchmarking testbed is to determine the placement of emulated WoT devices on the multicore hardware. Specifically, a deployment plan is needed that efficiently uses the capacity of the emulation testbed and decides

the placement of emulated devices on the processing cores. The second challenge is to design an approach for contention detection to ensure the accuracy of test results. While the deployment plan can reduce the effect of resource contention, it cannot guarantee to avoid it. Therefore, users of the benchmarking testbed should be aware of resource contention. Operating System (OS) level resource utilization metrics are not always sufficient to detect all sources of contention [46]. Furthermore, it might be infeasible to use hardware counter data for detecting contention since collecting these metrics often requires considerable processing resources [60]. As a result, developing an explicit approach to detect contention can help users of the emulation testbed by warning them about the potential effect of resource contention on test results.

This paper addresses the following research questions related to these challenges:

- **RQ 1**: How to create a scalable benchmarking testbed that emulates the application behaviour of multiple WoT devices having small resource footprints?
- **RQ 2**: How can commodity multicore hardware be used to host the testbed?
- **RQ 3**: How can contention for the shared resources be identified in the testbed?

In this paper we develop and present a novel WoT benchmarking solution: **WoTbench**, a **W**eb **o**f **T**hings **bench**marking testbed. To answer **RQ 1**, WoTbench is designed and implemented using Linux container technology. The testbed supports the emulation of a WoT system where the Constrained Application Protocol (CoAP) [55] is used as the application layer protocol. WoTbench employs containerized WoT device emulators that execute CoAP on Linux. The use of containers allows one to configure the usage characteristics of the various low level resources, e.g., sensors, network interface, and CPU core, used by a device as well as the conditions experienced by the WoT network, e.g., packet losses and delay. WoTbench also provides a synthetic workload generator that offers control over CoAP request arrival patterns. WoTbench allows these system and workload characteristics to be varied systematically as part of performance evaluation exercises.

RQ 2 is answered by enumerating and examining different considerations regarding the placement of different components of WoTbench on a multicore server. The common goal of these considerations is to increase the number of devices that can be deployed on a particular multicore hardware while reducing the possibility of resource contention. In the paper, we also use example results from a case-study to illustrate the value of workload and architecture dependent considerations, including the choice whether to use CPU affinity to bind different containers to particular cores or not.

To address **RQ 3**, we design and implement a Contention Detection (CD) module that flags resource contention in the underlying platform in any given test executed on WoTbench. This lightweight module consists of a custom emulated WoT device and a load generation application. The resource usage characteristics of the custom device are automatically derived based on the workload characteristics of the planned test. We show how the response time metrics derived from the emulated device can be used to infer shared resource contention that can potentially influence the accuracy of performance results reported by the test.

We make the following two main contributions with this work:
- We develop WoTbench[1], a toolkit that leverages libCoAP [34] and Docker [44] containers to enable a WoT testbed to be quickly created on multicore servers in a LAN environment.
- We implement the CD module to detect resource contention in the testbed, enabling more accurate assessment of the capabilities of the system under test. We show that the approach is effective in detecting contention in different scenarios.

To the best of our knowledge, other emulation frameworks are not contention aware and no single framework supports all the features implemented by WoTbench.

This paper is a major extension of our preliminary work, a 4-page short paper [30], which presented a high-level architecture of WoTbench. However, the architecture proposed in that work did not focus on contention awareness. Furthermore, in contrast to this paper, the prior work did not experimentally evaluate WoTbench.

The rest of this paper is organized as follows. Section 2 provides background information on WoT and discusses related work on WoT benchmarking. Section 3 motivates WoTbench by describing a capacity planning use case. Section 4 presents an overview of WoTbench and its components. Section 5 enumerates the main considerations when deploying WoTbench. Section 6 characterizes the impact of contention on test results and discusses how the CD module ensures the integrity of tests in the presence of such contention. Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 The Web of Things

It is estimated that 7 to 10 Billion IoT devices were active at the beginning of 2019 [10] [13]. The majority of these devices (*e.g.,* smart appliances, home security systems) are being monitored and controlled by users who usually prefer simplicity of access to these devices. Moreover, to improve usability and utility, IoT devices require a standardized way of communicating with each other. Web-enabling IoT devices allows for the use of existing infrastructures and eliminates the need for new client applications (other than the Web browser) to communicate with these devices.

There are several challenges involved in connecting IoT devices to the Web. For example, there is a need for revisiting standards, security and performance [62]. While there is a large body of ongoing studies focused on addressing standards and security challenges in WoT, less work has been devoted to developing tools to assess the performance and scalability of a network of WoT devices under design. We focus on this aspect in our study.

2.2 Constrained Application Protocol (CoAP)

A fundamental limitation in most WoT systems is the limited processing capability and power constraints of devices. CoAP [55], designed by the IETF CoRE Working Group [12], considers this limitation and provides a lightweight approach for connecting devices to the Web. It is designed to operate with a RESTful [24] architecture that results in a stateless nature. This allows the development of uniform HTTP-CoAP proxies to integrate devices with CoAP support to the Web. Similar to HTTP, in CoAP each device can have

---

[1] https://github.com/RSH2000/WOTbench

multiple resources that each have a unique Universal Resource Identifier (URI). A URI can be used to access a resource by sending GET, PUT, POST and DELETE requests. The resources in a WoT environment are typically the methods for reading data or modifying the settings of devices, sensors, actuators and communication mediums in the network. In contrast to HTTP, CoAP adopts UDP as the transport layer protocol considering the resource constrained nature of devices.

Since the initial design of CoAP in 2011, several implementations [11] have been introduced that are intended for different OS and hardware architectures. One of the widely used implementations is libCoAP [34]. It is written in C and can run on constrained device OSs (i.e., Contiki [20], TinyOS [37], RIOT [15]) as well as larger POSIX [8] OSs. Californium [33] in another popular implementation of CoAP, written in Java. It mainly targets back-end services and server nodes communicating with constrained devices. However, it can also be used on more powerful IoT nodes. Other examples of CoAP implementations include CoAPthon [58] in Python and node-coap [5] in JavaScript.

2.3 WoT Performance Evaluation Frameworks

This section describes some examples of existing solutions for evaluating the performance and scalability of WoT environments. The solutions available depend on the type of application or devices under test, the stage of development and the technology (e.g., protocols, algorithms, or applications) that needs to be emulated. This section categorizes the solutions into four different groups, namely, real sensor node testbeds, simulation and emulation solutions, protocol proxy and gateway applications, and synthetic traffic generators.

Real Sensor Node Testbeds: In recent years, several real experimental facilities were developed to facilitate testing and scalability analysis of a network of devices in the design and implementation [25, 54] stages. Some examples of such testbeds are IoT-lab [50], Smartsantander [53], WISEBED [19] and WoTT [16]. The first three testbeds are mainly developed for general IoT systems. However, they can be used for testing WoT applications. As an example, Paganelli et al. [49] used the Smartsantander [53] testbed to test their WoT solutions.

WoTT [16] is a testbed specifically designed for WoT applications. WoTT supports two node types, namely, constrained-IoT with power limitations and a Single Board Computer that is a more powerful node. WoTT supports multiple protocols, frameworks and platforms. In particular, it supports CoAP as an application layer protocol for constrained nodes. The testbed is part of the IoT-Lab project and enables testing of development features, such as service discovery, human interactions and user localization [9]. More recently, the Fiesta-IoT [35] project was established with the goal of creating a federation of existing real IoT testbeds to provide experimentation as a service solutions.

Using real hardware may result in more accurate evaluation of application performance. However, users can potentially face problems such as a limited number of nodes, lack of realistic network characteristics and the need for external workload generation tools. The high cost, limited scale, and lack of flexibility of real IoT setups motivate the need for alternative simulation or emulation techniques such as WoTbench.

Simulators and Emulators: A large number of frameworks have been developed to simulate IoT [2, 22, 36, 48] and in particular WoT [18, 21, 42] environments. Simulators such as Cooja [48] and TOSSIM [36] are specifically used for simulating the behaviour of IoT applications running on constrained device OSs. Cooja/MSPSim [22] further extends the features of Cooja and combines it with a hardware simulator (i.e., MSPSim) to enable white-box performance testing of IoT applications on their intended simulated hardware. Brambilla et al. [18] proposed a simulation methodology focused on urban IoT environments with an application layer perspective. Their approach provides support for simulating CoAP.

D'Angelo et al. [21] studied a multi-layer approach to simulate large scale IoT systems. Their approach is based on a coarse grained agent-based simulation followed by a fine-grained approach to simulate wireless communication links. Although their approach supports multiple communication strategies, it does not support simulation of RESTful application layer protocols such as COAP. Younan et al. [61] proposed a WoT-specific testbed architecture that focuses on smart home applications.

Simulation approaches can suffer from a lack of representativeness compared to real deployments. Moreover, simulation solutions typically have limited support to simulate sensor nodes with heterogeneity in features and communication protocols [32]. Emulation frameworks address some of these limitations. Specifically, emulation approaches can potentially allow the execution of parts of actual applications and protocols. Several IoT-specific emulation frameworks have been developed recently [27, 38, 43, 51] as we describe next.

MAMMoTH [38] is a large scale IoT emulator with support for emulating mobile device, wireless sensor networks and constrained devices. It reuses existing network simulation solutions (e.g., ns3 [6]) to simulate the behaviour of IoT networks and can execute CoAP as an application layer protocol. EmuFog [43] is an emulation framework for Fog computing [17] environments. It supports the use of existing network topology generators to emulate the environment. Similar to WoTbench, EmuFog uses Docker to enable evaluation of Fog nodes in the emulated network environment. However, it does not support CoAP.

Protocol Proxy and Gateway Applications: The third group of tools used for performance evaluation studies of IoT and WoT environments are gateway and proxy applications. These tools do not provide a comprehensive testbed to test a protocol or an application. However, they can facilitate the performance evaluation process. For example, Ponte [7] is a multi-transport IoT broker tool developed as part of the the Eclipse project. The tool functions as a proxy that supports messages from multiple IoT application layer protocols (CoAP, MQTT, HTTP) and can be used by developers to implement a multi-protocol transport module for their application. Ludovici et al. [40] designed an HTTP-CoAP proxy that facilitates the integration of a CoAP-based WoT environment with existing Web services. Their tool can be coupled with existing Web benchmarks to enable performance evaluation of WoT applications.

Workload Characterization and Synthetic Traffic Generators: IoT and in particular WoT use cases are relatively new. Moreover, due to the heterogeneity of systems and use cases, and privacy issues that limit data sharing, there are very few real, reusable workload traces. Therefore, synthetic traffic generators are needed to

facilitate performance evaluation. As an example, IoTAbBench [14] is a synthetic workload generation tool based on Markov-chains to emulate the behaviour of smart meters. Another example of such a tool is RIoTBench [56], a benchmark suite for evaluating distributed stream processing systems for IoT applications. It uses a combination of a set of micro benchmarks and four real-world data streams to build a set of four representative stream processing applications for IoT environments. More specific to WoT environments, CoAP-bench is a load generator tool which is part of the Californium [33] framework. It mimics the features of Apachebench [1] for the CoAP protocol. However, unlike WoTbench, it does not have support for generating traces with specified distribution of workload characteristics (e.g., request type and resource distribution).

Despite these advances, existing emulation approaches do not yet provide an integrated mechanism to evaluate how the performance of CoAP-based systems can change as a function of the overall system architecture, e.g., device and gateway topology, application characteristics, CoAP request arrival patterns, device resource demand distributions, and network conditions. Furthermore, in contrast to WoTbench, they do not consider the effect of test platform resource contention on the integrity of test results.

3 USE CASE SCENARIO

A WoT environment consists of several devices, typically with power constraints. These WoT devices communicate with one or more (typically more powerful) gateway nodes that connects them with the outside world. One approach to benchmark a WoT system is to deploy the gateway and devices in a real testbed. As discussed previously, real testbeds consisting of actual WoT sensor nodes may result in a more representative benchmarking experience and enable testing of already implemented applications prior to deployment. However, they are costly and may not be always available at the desired scale [32]. Emulation testbeds facilitate the benchmarking process in the design phase and defer the use of real hardware to after implementation. This section further explains the use case for a CoAP enabled emulation testbed.

CoAP/HTTP proxies allow Web users to access Web services provided by constrained devices through Web browsers. In these cases, the CoAP part of this communication may remain transparent from the users' perspective [59]. Figure 1 shows a simplified architecture of such a service. In this case, the CoAP devices can be temperature or air quality sensors in different locations. The user request may involve reading one or more sensors for a single point or a historical trend.

Similar to conventional Web services, capacity planning exercises are needed to ensure that these WoT applications provide acceptable experience to an end user, e.g., fast responses to sensor data requests. Typically, capacity planning is required for the gateway, proxy, and the device tiers. A common capacity planning approach is to answer what-if questions such as the following:

(1) For a given number of devices, an expected user behaviour, i.e., workload, and specific set of resources, e.g., sensors and their read service times, what is the maximum rate at which the gateway can read the sensor data while satisfying a desired response time?

(2) How do alternative implementations of application level protocols, e.g., congestion control and packet recovery protocols, layered on top of CoAP compare in terms of performance?
(3) What is the impact of having heterogeneous devices with varying computational capabilities?
(4) What is the effect of network characteristics such as individual device bandwidths, WoT network packet loss, delay and jitter on the maximum sensor read rate?
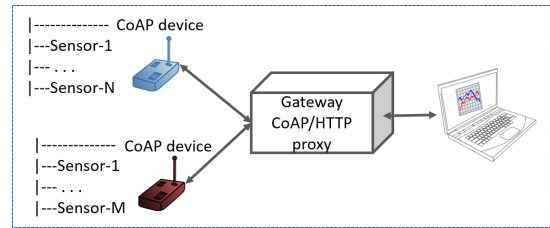

Figure 1: An example architecture of Web services in WoT environments.

WoTbench provides a testbed to facilitate experiments to answer these type of questions. It uses a Docker container to emulate a WoT-device. A WoT-device can exchange messages with the gateway using the actual CoAP protocol executing on Linux. Furthermore, WoTbench can emulate synthetic resources, e.g., sensors, attached to the WoT-device. It allows control over the resource demand distributions of these synthetic resources as well as the fraction of the testbed's computational and networking resources allocated to each device. The testbed also supports control over the pattern of CoAP request arrivals from the gateway to any given WoT-device.

The ability to specify request arrival patterns and resource demand distributions allows one to answer the first question in the sample capacity planning study. The ability to execute CoAP allows the evaluation of alternative application layer protocols as part of the second question. To answer the third question, the heterogeneity of the devices can be reflected by appropriately configuring the distribution of resource demands per WoT-device and by using the CPU and network sharing mechanisms supported by Docker [3]. For example, a device with high computation capability can be emulated by assigning to it a large fraction of the testbed's CPU resources. Finally, to answer the last question, WoTbench supports integration of an existing network emulator [23] to systematically perturb characteristics such as packet loss and delay on a per WoT-device basis.

4 WOTBENCH ARCHITECTURE

4.1 Overview

An overview of the WoTbench architecture is presented in Figure 2. WoTbench consists of four main components, namely, the WoTbench core, the Gateway Emulator (GE), WoT-device and the Contention Detection (CD) module. In addition to those components, the WoTbench environment can accommodate an existing network emulation tool called Pumba [23] to apply the expected network characteristics of the deployment environment. Except for the WoTbench core that is a process running on the platform's

OS, all of the other components consist of one or more Docker [44] containers connected through a virtual bridge network.
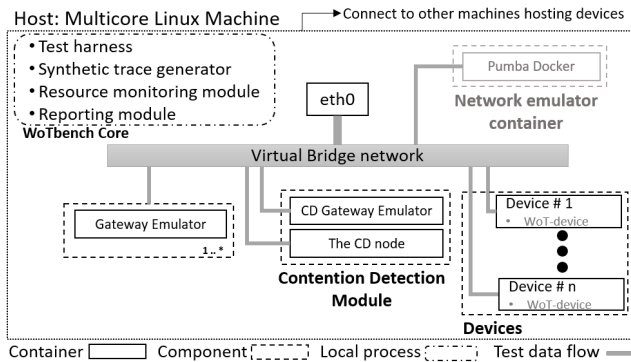

Figure 2: The WoTbench architecture.

WoTbench reports the per request response time measured at the GE as the main performance metric collected during an experiment. The response time is measured by subtracting the request sent timestamp from the response received timestamp. As a result, the reported response time includes the network transfer time.

4.2 The WoTbench Core

The core component of WoTbench consists of several processes that run in offline or online mode when conducting a test. Specifically, the WoTbench core consists of the following components:

- **Test harness**: This is the main process that controls other components and processes of WoTbench. This process is a shell script that performs multiple tests with specific characteristics. The test harness creates the environment (e.g., starts the Dockers) based on the test specifications, initiates the test, waits for the test to finish and finally collects the results and cleans the environment (e.g., stopping Dockers).
- **Synthetic trace generator**: This is an optional component, used to create synthetic workloads with specific service time distributions for the synthetic device resources and Request Inter-arrival Time (*RIT*) distributions for devices. These distributions can match service times and arrival patterns observed in an actual deployment or can be varied as part of a sensitivity analysis. The process runs in offline mode prior to the start of each test to generate a trace of requests for each device. The generated workload consists of a set of trace files and a group of resource files describing the set of synthetic resources and their service times for each device.The traces are then combined and fed to the GE to submit the workload. In contrast with most synthetic Web workload generators that use a probabilistic Finite State Machine (FSM) to represent user behaviour, WoTbench's trace generator selects the resources to access in a device with the goal to satisfy a desired distribution of service times for each device.
- **Resource monitor**: This module is a wrapper script around the Collectl [52] performance monitoring tool. It runs in

online mode to collect resource usage metrics (e.g., CPU utilization, memory usage) of the underlying hardware. The information can be used in conjunction with the CD module to investigate possible root causes of contention in the benchmark setup.

- **Reporting module**: This module generates a summary of test results and visualizes the relationship between the workload characteristics and the collected performance metrics.

4.3 Gateway Emulator

The Gateway Emulator (GE) plays the role of a workload generator [28] for the WoT devices, similar to tools such as httperf [45] or Apache JMeter [26] in a traditional Web benchmarking setup. It sends requests to each of the devices based on the input workload trace and measures each device's response time. Figure 3 depicts the process of submitting the workload to the devices by the GE. The GE is an asynchronous but single threaded process. It consists of a single busy loop for sending requests based on the send timestamps, specified by the workload trace. As a result, it requires a dedicated CPU core. WoTbench's test harness uses Docker's core affinity [39] feature to pin the GE process to a single core. Furthermore, it ensures that no device is emulated on the core that is running the GE.

The GE reads the workload trace from a *csv* file provided as input. The following is an example of a request in the workload file:

0, 1000, GET, coap://172.18.0.60/resource123

In this case, 0 is the device id, 1000 is the timestamp (representing microseconds from the start of the test) for sending that request, *GET* is the request type and *coap://172.18.0.60/resource123* is the URI that is to be accessed by this request. The workload can be specified based on traces collected from a real deployment or using a synthetic trace generated by the synthetic trace generator. The latter has the additional feature of generating a custom set of synthetic resources to satisfy a desired device service time distribution, as discussed previously.

The GE currently only supports CoAP. In real deployments, certain devices might have the capability to support HTTP. However, adding HTTP support is deferred to future work. The GE leverages libCoAP [34] to communicate via CoAP. libCoAP supports constrained OSs such as Contiki [20] and TinyOS [37], and also supports POSIX environments, where the emulated devices of WoTbench are expected to be deployed.

The overall process of sending a request and receiving the response works as follows. The GE first reads a workload trace from a file and initializes a timer to track the request send time. In each loop iteration, the timer's elapsed time is compared with the next request's send timestamp. If the next send timestamp is reached, the GE sends the next request; otherwise, it checks if any reply has arrived from the device via the non-blocking Unix *Select()* function. If no reply is waiting and there are more requests left, the loop continues; otherwise, the reply is processed. The GE then continues from the beginning of the loop to send the next request. The scalability and accuracy of the GE is further discussed in Section 6.
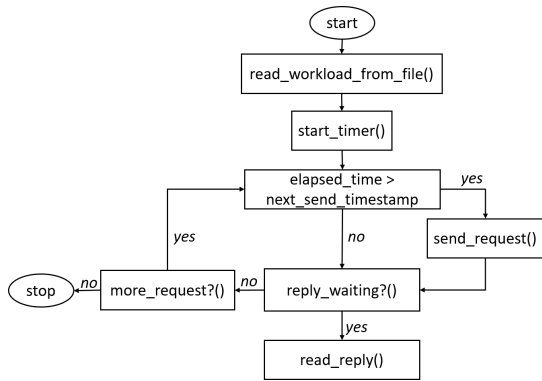
Figure 3: Gateway Emulator request generation process.

4.4 Devices

The devices are emulated as Docker containers. As introduced previously, WoTbench provides a specific implementation of a CoAP device, referred to as the WoT-device. The WoT-device is a multithreaded version of libCoAP server [34] running in a Docker container. The number of server threads, the emulated device resources and their attributes along with the expected service time distribution of the resources is configurable for each WoT-device in the test environment. The service time specifications can be gathered from initial prototypes or constructed based on sensor spec sheets.

The WoT-device emulates the service time in either the *sleep* or *busy* mode. The *sleep* mode uses the Unix *nanosleep()* [4] to emulate the service time, while the *busy* mode is a controlled CPU intensive loop that runs for the length of the service time. The two modes are used to emulate non CPU-intensive, i.e., sensor-intensive, and CPU-intensive CoAP requests, respectively.

4.5 The Contention Detection Module

The main role of the CD module is to monitor the testbed while a test is running and report if contention for the shared resources in the underlying multicore platform may have influenced the test results. The CD consists of two main components, an extra GE instance,[2] referred to as the Contention Detection GE (CD-GE) and an extra WoT-device node, referred to as the CD node. The CD-GE submits a sequential workload with a deterministic service time to the CD node and measures the response time. The CD then reports any response times that deviate from the deterministic service time, which may be an indicator of resource contention in WoTbench's multicore platform. Section 6.2 elaborates on the effect of contention on test results and describes the design and evaluation of the CD module in greater detail.

---

[2]Note that this is a separate instance from the GE used to submit the workload to the devices under test. The CD is an entirely independent part of the WoTbench infrastructure.

5 DEPLOYMENT PROCESS AND CONSIDERATIONS

The first step in creating a WoTbench deployment is to select the hardware platform. For instance, for this example, and as the deployment used in the experiments of the following sections, a single server is used with two processors, each having four cores. While WoTbench can be deployed on multiple machines, this evaluation focuses on a single machine deployment. This single machine platform was sufficient to run the chosen use case. The deployment process with multiple machines is left as future work.

Once the platform is selected, the next step is to design and apply the benchmark configuration. The benchmark configuration specifies the number of devices of different types and how the devices are positioned on the hardware platform. An ideal benchmark configuration should utilize the resources of the underlying platform as much as possible. Meanwhile, it should minimize the impact of contention for resources in the underlying hardware on the test results, which can adversely impact the integrity of test results. There are several design decisions that can help this process.

Figure 4 shows a carefully crafted example benchmark configuration with 10 devices for the above described platform. This benchmark configuration is created based on the following set of design considerations.

One example consideration is a desire to avoid contention between the OS processes and the WoTbench processes. Since core 0 normally is used to run OS processes [29], one should avoid scheduling the GE and device nodes on core 0. Instead, core 0 is used to run the test harness process which is not latency-critical. There are two GEs in this setup; one is used as the main workload generator for the test and a second is used by the CD module. The GEs each require a complete processing core. For this reason, as shown in Figure 4, no other container is scheduled on cores 2 and 4. Note that each complete WoTbench deployment requires at least a four core machine as the test platform. In the case of the eight core server used for this study, the other five cores are used to run the WoT-device nodes alongside the CD node.
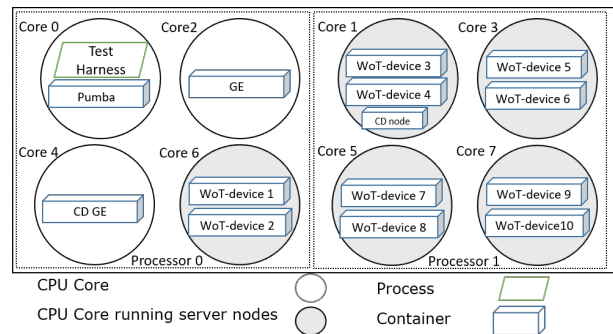

Figure 4: Benchmark configuration.

Another consideration is regarding the scheduling mechanism of devices on CPU cores. In particular, one needs to decide about whether to bind each container to a single core using CPU affinity [39], or alternatively, leave the container scheduling decisions to the Linux scheduler. Optimizing this decision has the advantage

of using the hardware resources more efficiently. The decision to use affinity depends on the workload and the types of devices under test. Therefore, the answer can be determined experimentally for each specific combination of device types and workload.

Next, an example of such an experimental process is presented for the above described platform. The goal is to measure the effect of CPU affinity on the capacity of a multicore server hosting WoTbench. The idea is to create a CPU intensive workload and determine which of the CPU scheduling choices can accommodate a larger number of devices without contention. The example workload considered for this study has exponentially distributed request inter-arrival and service times. The mean Request Inter-arrival Time (RIT) is set to 100 milliseconds per device while the mean service time is set to 16 milliseconds. The number of devices running on a single CPU core is then increased to measure the capacity of the multicore server for benchmarking, with and without core affinity.

Figures 5(a) and 5(b) show the measured mean response time and CPU utilization, respectively, as a function of the number of devices. Figure 5(a) shows that the use of CPU affinity had less than a 5% impact on the measured response time when less than 15 WoT devices (3 devices per core) are used. However, the measured response time increases for the tests with more than 3 devices per core. The total CPU utilization of the five cores as a function of the number of devices is depicted in Figure 5(b). This graph shows that the overall CPU utilization is less than 5% higher for the case without affinity. While the use of CPU affinity resulted in a slightly lower CPU utilization, it had an adverse effect on the response time, therefore, using affinity is not helpful in this case.

While for this particular scenario the use of CPU affinity was not helpful, the configuration can be beneficial in other cases such as deployments with one device per core or cases with heterogeneous devices. As a result, the initial decision around container placement and the use of affinity in those scenarios will be workload dependent. Moreover, despite the workload-based optimization of container placement, the effect of the communication channel and caching hierarchy, the position of the cores hosting the devices on the two processors can potentially affect the test results [29]. We currently ignore such effects at the deployment phase since the CD component is designed to detect them, as discussed in Section 6.2.

6 RESULT INTEGRITY

An important consideration when using any benchmarking system is to examine the integrity of the test results [31]. Specifically, a comprehensive benchmarking system should provide feedback to the benchmark user in case of deviations between the expected test configuration and the executed test. One example of such deviations happens when a bottleneck in the load generation system of an interactive benchmark results in the request inter-arrival times of the submitted workloads being inflated beyond what is specified for the input, i.e., the expected workload [28]. WoTbench addresses this scenario by reporting the timestamp of the sent requests and statistical characteristics of the submitted workload. Section 6.1 presents an example that elaborates on how the user can interpret this report to ensure the integrity of the generated workload.

Another source of deviations, more specific to WoTbench, is the resource contention in the platform hosting the benchmark. Resource contention is typically negligible in low load scenarios. However, at higher load levels, contention for shared resources of the underlying multicore hardware can adversely affect the results of tests running on WoTbench. The CD module helps identify if and at what loads such problems start to occur for a particular architecture and test setup. Section 6.2 describes and evaluates the effectiveness of this module, and discusses how the module can help WoTbench users safely scale their experiments.

6.1 Gateway Emulator Bottleneck

As mentioned in Section 5, each instance of the GE runs on a separate processing core of the underlying platform. The scalability of the GE depends on the specification of the hardware platform. The capacity of the GE on a specific platform is determined by the minimum time it takes for the GE to submit two consecutive requests. This limits the minimum Request Inter-arrival Time (RIT) that the GE can emulate.

To elaborate on this limitation, the WoTbench deployment described in Section 5 is used to run a test with a sample workload in which the RIT from the GE perspective is exponentially distributed with mean of 300 microseconds (or 3,333 requests per second on average). The test is run twice; once when there is a single device and again with ten devices under test. Note that the expected RIT distribution from the GE perspective is the same for the two scenarios. This means that the average RIT per device is ten times higher for the test with ten devices. Figure 6 visualizes the Cumulative Distribution Function (CDF) of RIT for the submitted and input (expected) workload for the single device case, which has the higher per-device request load (*i.e.*, small average per-device RITs). Note that the zoomed in area of the graph shows a clear separation between the 10% of requests with the smallest RITs. In particular, for the submitted workload, this subset of low-RIT values all have RITs of approximately 30 microseconds, whereas the expected workload includes much lower RITs. This confirms that the GE was not able to submit the requests with an inter-arrival time of less than 30 microseconds. The effect of this limitation on the integrity of the results depends on the per-device workload and is determined by comparing the statistical characteristics of expected and submitted workloads, as reported by the GE.

Table 1 shows the reported values for mean and 5th percentiles of the RIT for the two tests with one and ten devices, respectively. These parameters are selected to show the effect of the minimum RIT on the lower percentile and the mean. The results show that the RIT distribution diverges more when the emulator is submitting its workload to a single device. The GE's limitation changed the mean RIT by 0.5% and 0.05% for the tests with 1 and 10 devices, respectively. The differences are even more clear when looking at the 5th percentile. For example, while the 5th percentile is doubled for the single device case, it remained unchanged for the ten device scenario. The data provided by the WoTbench GE can be used in this manner for evaluating the correctness of tests with arbitrarily specified RIT distributions.
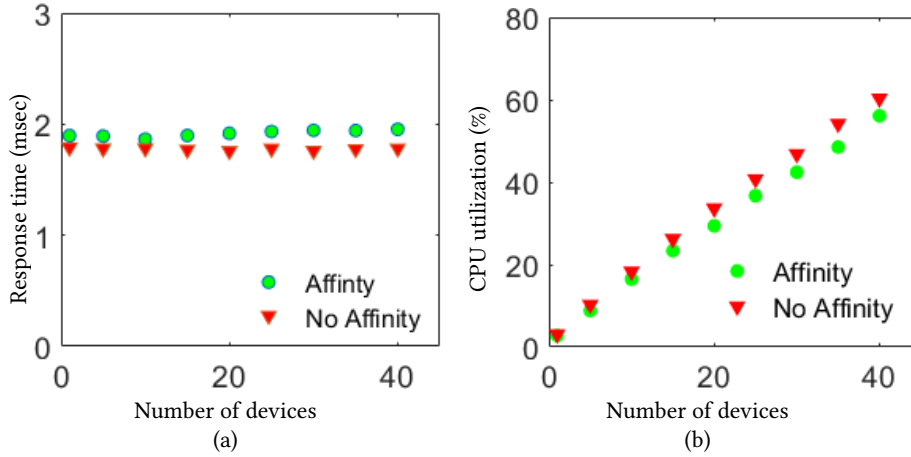
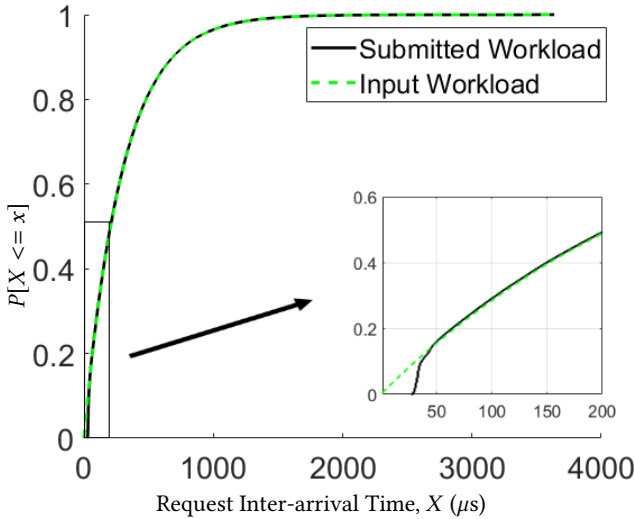Figure 5: Examining CPU affinity: (a) Response time and (b) CPU utilization.

Figure 6: Cumulative Distribution Function of *RIT* for test with high request load (*i.e.*, small RITs) and a single device.

Table 1: Submitted vs. Expected workload.

Number of devices	RIT values (μs)			
	Input Workload		Submitted Workload	
	Mean	5th %ile	Mean	5th %ile
1	300	15	286	32
10	3,019	150	3,005	150

In summary, WoTbench provides a tool for users to extract and compare reported *RIT* values, and to determine an appropriate number of GE instances. In particular, if the WoTbench user finds that the mean or percentile deviations are not acceptable, one or more additional GE instances have to be used to submit the workload.

6.2 Contention Detection Module

A threat to the integrity of tests running on WoTbench is contention for shared resources. As discussed in Section 5, while a well designed benchmark configuration can reduce the probability of such contention, it does not guarantee its absence.

The contention problem: To understand the implications of contention, three benchmark configurations are examined in which 1, 5 and 20 homogeneous WoT-device nodes each subjected to the same workload are deployed using the platform described in Section 5. In all cases, the device's service time distribution is exponential with a mean of 5 milliseconds. Moreover, the WoT-devices are running in busy mode to emulate a CPU-intensive workload. If there were no contention in the platform, the response time characteristics at any given per-device throughput should be the same across these three configurations.

Figure 7(a) shows the response times, as a function of per device request throughput. Note that the response time is similar for the three scenarios up to the per device throughput of 40 requests per second. For higher throughputs, the measured response times are higher for the 20 device scenario compared to the 1 and 5 device cases. Considering that the device service time distributions and the *RIT* distributions are identical for the three test scenarios, the inflated response time in the 20 devices case can be an indicator of resource contention at the platform level. The results suggest that the testbed does not have the capacity of hosting 20 devices under this specific workload.

While the existence of resource contention is clearly observable in this controlled experiment, this effect cannot be easily detected in a real test scenario. In testbeds with multiple heterogeneous devices and variable workload conditions, testing such controlled scenarios typically requires running a large number of experiments and a lengthy benchmarking process. Furthermore, resource utilization and OS level metrics are not always sufficient to detect all sources of contention. For example, if the contention happens in parts of the CPU or memory subsystem (e.g., cache hierarchy), hardware level metrics are required to analyze the contention scenario. However, collecting and analyzing these metrics is costly and requires
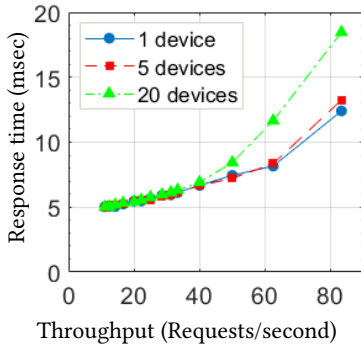
Figure 7: Effect of platform resource contention on measured response time

Figure 8: CD node's average response time for contention scenario.

considerable processing resources and is therefore not feasible in most use cases. As a result, an alternative approach is required to ensure the correctness of test results while eliminating the need for running multiple experiments.

Our CD module solution: The proposed solution within WoTbench is the CD module. The method adds a single WoT-device, i.e., CD node, and an extra Gateway Emulator container, i.e., Contention Detector GE (CD-GE), to the WoTbench deployment. During the experiment, the CD-GE submits a controlled workload to the CD node and measures its response time. The basic idea is that when there is no contention in the underlying hardware, the CD node's response time is close to its configured service time. However, when the device nodes suffer contention for a shared resource, the CD node will experience this contention as well. Consequently, the measured response time of the CD node will deviate from the configured service time, i.e., the wait time for the congested shared resources is added to the service time.

There are two main characteristics for the CD workload. First, requests should arrive sequentially with deterministic inter-arrival times to ensure that the requests are not queued for the software resources of the CD node. Second, the CD node's resources should have deterministic service times. These two conditions ensure that any deviation between response time and configured CD node service time can be attributed to resource contention.

Returning to the contention example above, we now explain how the CD module can help detect resource contention. Figure 8 shows the CD node's average response time as a function of per device throughput for the same experiments described above. For these experiments, the CD node requests are sent with the constant inter-arrival time of 1 second and the service time of 5 milliseconds, which is the same as that used by the devices. From the figure, the average response time reported by the CD node was almost constant for the 1 device and 5 devices experiments. However, for the case with 20 devices, the average response time of the CD node increases from 5 milliseconds, i.e., the configured service time, to 8 milliseconds for the tests with a per device throughput of higher than 40 requests per second. Recall from Figure 7 that the response time inflation was observed for the same range of throughputs for the devices. This indicates that the CD node response time follows
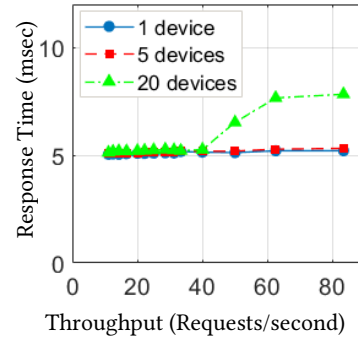
the same pattern observed for device nodes and can potentially thus detect such contention scenario in uncontrolled setups.

To quantify the extent of contention, WoTbench reports the normalized response time deviations $R_d = \frac{R-S}{S}$, where R is the response time and S is the service time. In practice, the response time also includes the network transfer time. Therefore, R_d is slightly greater than 0 even under no contention.

In the above contention example, the CD node was tuned based on the knowledge about the workload and the devices under test. In particular, the service time of the CD node requests are chosen to be the same as the average service time of the devices. Since tests use synthetic WoT-devices, the mean service time of the devices can be calculated from the service times specified as input. The CD node can then be configured with the estimated service time.

Next, additional experiments are presented to show the effectiveness of the CD module in detecting contention under different load conditions. First, WoT-devices are configured to have a mean service time of 25 milliseconds and operate in busy mode to create a CPU intensive workload. In the second case, the WoT-devices have the same service time but operate in sleep mode to emulate a non-CPU intensive workload. The mean service time of the CD node is set to be the same as the service time of the devices.

Figures 9(a) and 9(b) show the normalized response time deviations (R_d) values as a function of the number of devices for both the WoT-devices and the CD node for the CPU intensive and non-CPU intensive workloads, respectively. Figure 9(a) shows that the R_d of the device is constant and around 0.05 from 1 to 20 devices for the CPU intensive case. With 25 devices, R_d increases to 0.07 and with 30 devices it is measured as 0.09. Similarly, the R_d for the CD node is constant for up to 20 devices and starts to increases from the test with 25 devices. This result shows that the CD node could detect the emergence of contention while emulating 25 devices.

Figure 9(b) shows a similar trend for the non-CPU intensive workload. In this case, the device nodes' normalized response time deviation, R_d, is constant and below 0.055 for up to 90 devices. For the experiment with 100 devices, R_d increases to 0.07 and with 130 devices, the R_d is measured at 0.125. The CD node is able to follow the devices' response times and detect the contention for the case of 100 devices.
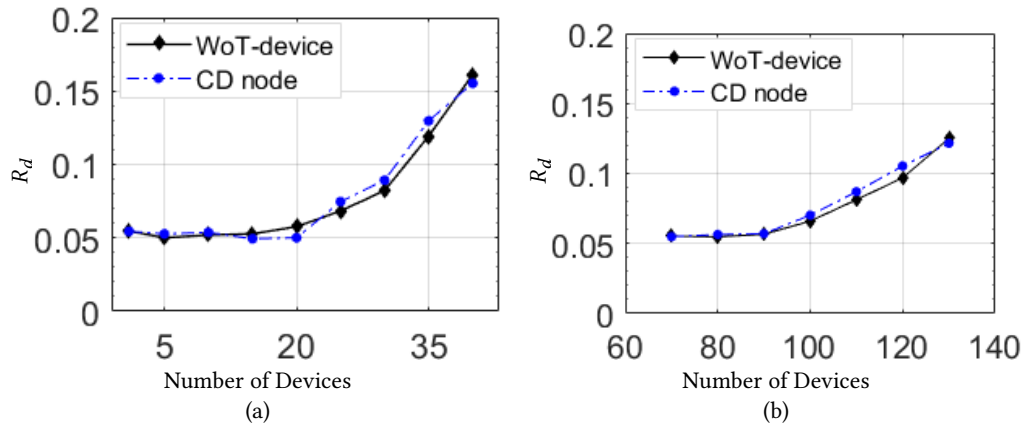
Figure 9: Normalized response time deviation (R_d) for WoT-devices and CD node : (a) CPU intensive workload, (b) Non-CPU intensive workload.

7 CONCLUSIONS

This paper describes WoTbench, an emulation testbed to facilitate benchmarking studies in WoT environments. WoTbench is designed to be deployed on commodity multicore hardware. It allows users to conduct capacity planning studies and examine behaviour of WoT systems under different system architectures, application characteristics, workload scenarios and network conditions. WoTbench's Contention Detection (CD) module also enables the users to carefully scale their experiments, through the monitoring and detection of potential resource contentions that can affect the test results.

Future work will focus on automating the deployment process described in Section 5. We will also expand the gateway emulator to support HTTP. Other interesting future work directions include the integration of real devices and applications and the ability to emulate power and energy usage characteristics of specific devices of interest.

REFERENCES

[1] [n.d.]. ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4. https://httpd.apache.org/docs/2.4/programs/ab.html. Last accessed: [2019-08-22].

[2] [n.d.]. IoT Analytics - ThingSpeak Internet of Things. https://thingspeak.com/. Last accessed: [2019-08-22].

[3] [n.d.]. Limit container resources. http://docs.docker.com/config/containers/resource_constraints. Last accessed: [2019-08-22].

[4] [n.d.]. nanosleep(2) - Linux manual page. http://man7.org/linux/man-pages/man2/nanosleep.2.html. Last accessed: [2019-08-22].

[5] [n.d.]. node-coap. https://github.com/mcollina/node-coap. Last accessed: [2019-08-22].

[6] [n.d.]. ns-3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/. Last accessed: [2019-08-22].

[7] [n.d.]. Ponte: Connecting Things to Developers. http://www.eclipse.org/ponte/. Last accessed: [2019-08-22].

[8] [n.d.]. Posix Standard - Linux Hint. https://linuxhint.com/posix-standard/. Last accessed: [2019-08-22].

[9] [n.d.]. WoTT - Internet of Things Laboratory. http://iotlab.unipr.it/wott/. Last accessed: [2019-08-22].

[10] 2018. State of the IoT 2018: Number of IoT devices now at 7B, Market accelerating. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/. Last accessed: [2019-08-22].

[11] 2019. CoAP: Constrained Application Protocol, Implementations. http://coap.technology/impls.html. Last accessed: [2019-08-22].

[12] 2019. Constrained RESTful Environments (core). https://datatracker.ietf.org/wg/core/charter/. Last accessed: [2019-08-22].

[13] 2019. Internet of Things Report: Technology Trends and Market Growth in 2019 - Business Insider. https://www.businessinsider.com/internet-of-things-report. Last accessed: [2019-08-22].

[14] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. 2015. IoTAbench: an internet of things analytics benchmark. In *Proc. of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 133–144.

[15] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE conference on computer communications workshops (INFOCOM WK-SHPS)*. IEEE, 79–80.

[16] Laura Belli, Simone Cirani, Luca Davoli, Andrea Gorrieri, Mirko Mancin, Marco Picone, and Gianluigi Ferrari. 2015. Design and Deployment of an IoT Application-Oriented Testbed. *Computer* 48 (09 2015), 32–40. https://doi.org/10.1109/MC.2015.253

[17] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*. Springer, 169–186.

[18] Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, and Francesco Zanichelli. 2014. A simulation platform for large-scale internet of things scenarios in urban environments. In *Proceedings of the First International Conference on IoT in Urban Space*. ICST (Institute for Computer Sciences, Social-Informatics and …, 50–55.

[19] Ioannis Chatzigiannakis, Stefan Fischer, Ch. Koninis, G. Mylonas, and D. Pfisterer. 2009. WISEBED: an open large-scale wireless sensor network testbed. In *International Conference on Sensor Applications, Experimentation and Logistics*. Springer, 68–87.

[20] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462.

[21] Gabriele D'Angelo, Stefano Ferretti, and Vittorio Ghini. 2017. Multi-level simulation of internet of things on smart territories. *Simulation Modelling Practice and Theory* 73 (2017), 3–21.

[22] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. 2009. COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques (Simutools '09)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, Article 27, 7 pages. https://doi.org/10.4108/ICST.SIMUTOOLS2009.5637

[23] Alexei Ledenev et. al. [n.d.]. Pumba: Chaos testing tool for Docker. https://github.com/alexei-led/pumba. Last accessed: [2019-08-22].

[24] Roy T Fielding and Richard N Taylor. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation.

[25] Alexander Gluhak, Srdjan Krco, Michele Nati, Dennis Pfisterer, Nathalie Mitton, and Tahiry Razafindralambo. 2011. A survey on facilities for experimental internet of things research. (2011).

[26] Emily H Halili. 2008. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd.

[27] Martin Grambow Elias Grünewald Sascha Huk Hasenburg, Jonathan and David Bermbach. 2019. MockFog: Emulating Fog Computing Infrastructure in the Cloud. In *Proc. of the First IEEE International Conference on Fog Computing*.

[28] Raoufehsadat Hashemian, Diwakar Krishnamurthy, and Martin Arlitt. 2012. Web Workload Generation Challenges - an Empirical Investigation. *Softw. Pract. Exper.* 42, 5 (May 2012), 629–647. https://doi.org/10.1002/spe.1093

[29] Raoufeh Hashemian, Diwakar Krishnamurthy, Martin Arlitt, and Niklas Carlsson. 2013. Improving the scalability of a multi-core web server. In *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 161–172.

[30] Raoufeh Hashemian, Diwakar Krishnamurthy, Niklas Carlsson, and Martin Arlitt. 2019. WoTbench: A Benchmarking Framework for the Web of Things. In *Proc. of the 9th ACM International Conference on the Internet of Things*. ACM, 1–4.

[31] R. Jain. 1991. *The Art of Computer Systems Performance Analysis*. Wiley & sons.

[32] Gabor Kecskemeti, Giuliano Casale, Devki Nandan Jha, Justin Lyon, and Rajiv Ranjan. 2017. Modelling and simulation challenges in internet of things. *IEEE cloud computing* 4, 1 (2017), 62–69.

[33] Matthias Kovatsch, Martin Lanter, and Zach Shelby. 2014. Californium: Scalable cloud services for the internet of things with coap. In *2014 International Conference on the Internet of Things (IOT)*. IEEE, 1–6.

[34] Koojana Kuladinithi, Olaf Bergmann, Thomas Pötsch, Markus Becker, and Carmelita Görg. 2011. Implementation of coap and its application in transport logistics. *Proc. IP+ SN, Chicago, IL, USA* (2011).

[35] Jorge Lanza, Luis Sanchez, Juan Ramon Santana, Rachit Agarwal, Nikolaos Kefalakis, Paul Grace, Tarek Elsaleh, Mengxuan Zhao, Elias Tragos, Hung Nguyen, et al. 2018. Experimentation as a service over semantically interoperable Internet of Things testbeds. *IEEE Access* 6 (2018), 51607–51625.

[36] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM, 126–137.

[37] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

[38] Vilen Looga, Zhonghong Ou, Yang Deng, and Antti Ylä-Jääski. 2012. Mammoth: A massive-scale emulation platform for internet of things. In *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, Vol. 3. IEEE, 1235–1239.

[39] R. Love. [n.d.]. CPU Affinity. http://www.linuxjournal.com/article/6799. Last accessed: [2019-08-22].

[40] Alessandro Ludovici and Anna Calveras. 2015. A proxy design to leverage the interconnection of coap wireless sensor networks with web applications. *Sensors* 15, 1 (2015), 1217–1244.

[41] Li Ma, Yang Yang, Zhaoming Qiu, Guotong Xie, Yue Pan, and Shengping Liu. 2006. Towards a complete OWL ontology benchmark. In *European Semantic Web Conference*. Springer, 125–139.

[42] Cristyan Manta-Caro and Juan M. Fernández-Luna. 2018. Modeling and Simulating the Web of Things from an Information Retrieval Perspective. *ACM Transactions on the Web (TWEB)* 12, 1 (2018), 6.

[43] Ruben Mayer, Graser Leon, Gupta Harshit, Saurez Enrique, and Ramachandran Umakishore. 2017. Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *2017 IEEE Fog World Congress (FWC)*. IEEE, 1–6.

[44] D. Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[45] D. Mosberger and T. Jin. 1998. httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.* 26 (Dec. 1998), 31–37. Issue 3.

[46] Joydeep Mukherjee, Diwakar Krishnamurthy, Jerry Rolia, and Chris Hyser. 2013. Resource contention detection and management for consolidated workloads. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 294–302.

[47] Joydeep Mukherjee, Diwakar Krishnamurthy, and Mea Wang. 2016. Subscriber-driven interference detection for cloud-based web services. *IEEE Transactions on Network and Service Management* 14, 1 (2016), 48–62.

[48] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. 2006. Cross-level sensor network simulation with cooja. In *First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*.

[49] Federica Paganelli, Stefano Turchi, and Dino Giuli. 2014. A web of things framework for restful applications and its experimentation in a smart city. *IEEE Systems Journal* 10, 4 (2014), 1412–1423.

[50] Georgios Z Papadopoulos, Julien Beaudaux, Antoine Gallais, Thomas Noel, and Guillaume Schreiner. 2013. Adding value to WSN simulation using the IoT-LAB experimental platform. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 485–490.

[51] Brian Ramprasad, Joydeep Mukherjee, and Marin Litoiu. 2018. A Smart Testing Framework for IoT Applications. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 252–257.

[52] Mark S. [n.d.]. collectl. http://collectl.sourceforge.net/Documentation.html. Last accessed: [2019-08-22].

[53] Luis Sanchez, José Antonio Galache, Veronica Gutierrez, Jose Manuel Hernandez, Jesús Bernat, Alex Gluhak, and Tomás Garcia. 2011. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *2011 Future Network & Mobile Summit*. IEEE, 1–8.

[54] Luis Sánchez, Jorge Lanza, Juan Santana, Rachit Agarwal, Pierre Raverdy, Tarek Elsaleh, Yasmin Fathy, SeungMyeong Jeong, Aris Dadoukis, Thanasis Korakis, et al. 2018. Federation of Internet of Things testbeds for the realization of a semantically-enabled multi-domain data marketplace. *Sensors* 18, 10 (2018), 3375.

[55] Z. Shelby, K. Hartke, and C. Bormann. 2014. *The Constrained Application Protocol (CoAP)*. RFC 7252.

[56] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4257.

[57] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. 2008. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, Vol. 8. 228.

[58] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2015. CoAPthon: Easy development of CoAP-based IoT applications with Python. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 63–68.

[59] Floris Van den Abeele, Enri Dalipi, Ingrid Moerman, Piet Demeester, and Jeroen Hoebeke. 2016. Improving user interactions with constrained devices in the web of things. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE, 153–158.

[60] Vincent M Weaver. 2015. Self-monitoring overhead of the Linux perf_ event performance counter interface. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 102–111.

[61] Sherif Khattab Younan, Mina and Reem Bahgat. 2015. An Integrated Testbed Environment for the Web of Things. *ICNS 2015* (2015), 83.

[62] Deze Zeng, Song Guo, and Zixue Cheng. 2011. The web of things: A survey. *JCM* 6, 6 (2011), 424–438.