WoTbench: A Benchmarking Framework for the Web of Things

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

The Web of Things (WoT) is a new paradigm resulting from the integration of the Web with the Internet of Things. With the emergence of WoT comes the need for benchmarking tools to aid in performance evaluation studies that can be used to guide actual deployments of WoT infrastructure. In this paper we introduce WoTbench, a Web benchmarking framework designed to facilitate pre-deployment performance evaluation studies of WoT setups. WoTbench enables workload emulation on a controlled test environment by taking advantage of Linux container technology. It supports the Constrained Application Protocol (CoAP), a RESTful application layer protocol that aims to replace HTTP for resource constrained devices. WoTbench consists of a scalable Gateway Emulator (GE) and a group of CoAP device application emulators. We describe components of WoTbench in detail and how WoTbench can be used to answer capacity planning questions. [1]

CCS CONCEPTS

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

KEYWORDS

Web of Things, CoAP, benchmarking, container technology, performance evaluation, scalability testing

ACM Reference Format:
Raoufeh Hashemian, Niklas Carlsson, Diwakar Krishnamurthy, and Martin Arlitt. 2019. WoTbench: A Benchmarking Framework for the Web of Things. In *Proceedings of The 9th International Conference on the Internet of Things (IoT '19).* ACM, New York, NY, USA, 5 pages. https://doi.org/xxx

[1] Authors' version of work produced with ACM permission. Not for redistribution.

IoT '19, October 22–25, 2019, Bilbao, Spain

ACM ISBN xxx…$15.00
https://doi.org/xxx

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 [31], Semantic Web [21], multicore Web infrastructure [14] and cloud computing [25]. The introduction of WoT marks the beginning of another important paradigm that requires new benchmarking tools.

In this paper we develop a novel WoT benchmarking solution: **WoTbench**, a **W**eb **o**f **T**hings **bench**marking framework. The framework leverages commodity multicore server hardware as a testbed for emulating the performance behaviour of a group of CoAP devices each with a small resource footprint. WoTbench employs containerized WoT device emulators that execute CoAP over 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 experienced by the devices.

Using WoTbench, one can study the impact of various system architectures, e.g., different device interconnection topologies. It also allows comparison of alternative protocol features, e.g., various application-level congestion control mechanisms. In contrast to existing WoT simulation and emulation frameworks [7, 19, 22], WoTbench provides an integrated mechanism to evaluate how the performance of large CoAP-based systems can change as a function of the overall system architecture, application policies, device resource demands, workload patterns, and network conditions.

The rest of this paper is organized as follows. Section 2 provides background on the CoAP. Section 3 describes a use case for WoTbench. Section 4 presents an overview of WoTbench and its components. Section 5 discusses the deployment process. Section 6 reviews related work while Section 7 concludes the paper.

2 CONSTRAINED APPLICATION PROTOCOL

A fundamental limitation in most WoT systems is the limited processing capability and power constraints of devices. CoAP [30], designed by the IETF CoRE Working Group [2], considers this limitation and provides a lightweight approach for connecting devices to the Web. It is designed to operate with a RESTful 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 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 have been introduced that are intended for different Operating Systems (OS) and hardware architectures. One of the widely used implementations is libCoAP [16]. It is written in C and can run on constrained device OSs (e.g., Contiki [10], TinyOS [17]) as well as larger POSIX based OSs. Californium [15] 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 [32] in Python or node-coap [4] in JavaScript.

3 USE CASE FOR WOT BENCHMARKING

As mentioned earlier, CoAP is designed to provide a RESTful interface for constrained devices. The 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 [33]. 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. The gateway component is typically a more powerful device compared to CoAP devices. The gateway includes a CoAP/HTTP proxy to enable communication between the CoAP devices and users.

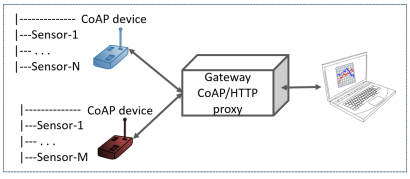

Figure 1: An example architecture of Web services in WoT environments.

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 policies, e.g., congestion control and packet recovery protocols, layered on top 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 this maximum rate?

WoTbench provides a testbed to facilitate experiments to answer these type of questions. It uses Docker containers [23] 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 level policies as part of the second question. To answer the third question, the heterogeneity of 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 [11] to systematically perturb characteristics such as packet loss and delay on a per WoT-device basis.

4 WOTBENCH

An overview of the WoTbench setup is presented in Figure 2. WoTbench consists of three main components, namely, the WoTbench core, the Gateway Emulator (GE), and WoT-devices. In addition to those components, the WoTbench environment can accommodate an existing network emulation tool called Pumba [11] 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 containers connected through a virtual bridge network.

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. We next

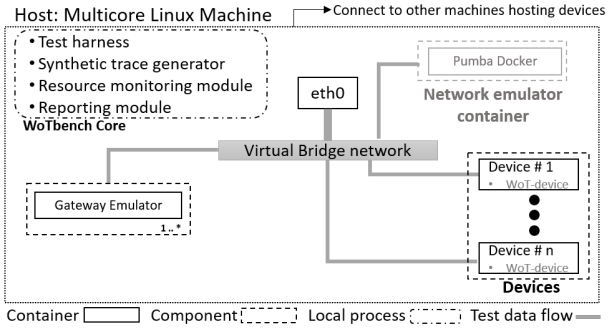

Figure 2: WoTbench architecture

describe the different components of WoTbench and the overall process of running a test with WoTbench.

4.1 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 device resources and request inter-arrival time distribution 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 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 [29] performance monitoring tool. It runs in online mode to collect resource usage metrics (e.g., CPU utilization, memory usage) of the underlying hardware.
- **Reporting module**: This module generates a summary of test results and visualizes the relationship between the workload characteristics and the collected performance metrics.

4.2 Gateway Emulator

The Gateway Emulator (GE) plays the role of a workload generator [13], similar to tools such as httperf [24] in a traditional Web benchmarking setup. It sends requests to each of the emulated devices based on the input workload trace and measures each device's response time. 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 [20] 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 using 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 service time distributions, as discussed previously. The GE currently only supports CoAP. In real deployments, certain devices might have the capability to support HTTP. However, we defer adding HTTP support to future work. WoTbench uses libCoAP [16] as the client library for CoAP communications.

4.3 Devices

The devices are emulated as Docker containers. WoTbench provides a specific implementation of a CoAP device, referred to as the WoT-device. The WoT-device is a multi-threaded version of libCoAP server [16] running in a Docker container. The number of threads, resources and their attributes along with the expected service time distribution of 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 service time in either the *sleep* or *busy* mode. The *sleep* mode uses the Unix *nanosleep()* [1] 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.

5 DEPLOYMENT PROCESS

The first step in creating a WoTbench deployment is to select the hardware platform. While WoTbench can be deployed on multiple machines, we focus on a single machine deployment. Once the platform is selected, we then need 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 optimal 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 optimize this process.

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 [14], we 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. Another consideration is to ensure a match of desired workload characteristics. To achieve this, each GE, i.e., the workload generator, in the setup needs a complete processing core. 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 [20], or alternatively, leave the container scheduling decisions to the Linux scheduler. The answer to this question should not affect the test results. However, 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.

6 RELATED WORK

In the past decade, real experimental facilities such as IoT-lab [27] and WoTT [6] were developed to enable testing and scalability analysis of a network of devices. Conducting a scalability test in a testbed with real devices may result in more accurate evaluation. However, the use of real testbeds might be not be feasible when evaluating large scale deployments. A large number of frameworks have been developed to simulate IoT [18, 26] and in particular WoT [8, 9] environments. Simulation approaches can suffer from a lack of representativeness to real deployments. Emulation frameworks address some of the limitations around the representativeness of simulation approaches. Specifically, emulation approaches can potentially allow the execution of parts of actual applications and protocols. Several IoT-specific frameworks have been developed recently [5, 12, 15, 19, 22, 28]. However, to the best of our knowledge, no one approach provides in an integrated way all the features supported by WoTbench.

7 CONCLUSIONS AND FUTURE WORK

In this paper we proposed WoTbench, a benchmarking framework for WoT environments. WoTbench can be deployed on commodity multicore hardware. It allows users to perform pre-deployment capacity planning studies of WoT systems. Future work includes applying WoTbench for testing a WoT environment with realistic test configurations. Moreover, we plan to add support for HTTP/CoAP conversion to the GE component. Finally, since the emulated WoT-devices run on a multicore platform, the contention for shared hardware resources may impact the test result. In future we plan to design an approach to detect these potential side effects on the test results.

8 ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] 2017. *nanosleep(2) - Linux manual page.* Retrieved 30/08/2019 from http://man7.org/linux/man-pages/man2/nanosleep.2.html
[2] 2019. *Constrained RESTful Environments (core).* Retrieved 30/08/2019 from https://datatracker.ietf.org/wg/core/charter/
[3] 2019. *Limit a container's resources.* Retrieved 30/08/2019 from https://docs.docker.com/config/containers/resource_constraints/
[4] 2019. *node-coap.* Retrieved 30/08/2019 from https://github.com/mcollina/node-coap
[5] F. Banaie, J. Misic, V. B Misic, M. H. Yaghmaee, and S.A. Hosseini. 2018. Performance Analysis of Multithreaded IoT Gateway. *Internet of Things Journal* (2018).
[6] L. Belli, S. Cirani, L. Davoli, A. Gorrieri, M. Mancin, M. Picone, and G. 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
[7] F. Bonomi, R. Milito, P. Natarajan, and J. 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.
[8] G. Brambilla, M. Picone, S. Cirani, M. Amoretti, and F. Zanichelli. 2014. A simulation platform for large-scale internet of things scenarios in urban environments. In *Proc. International Conference on IoT in Urban Space*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 50–55.
[9] G. DÀngelo, S. Ferretti, and V. Ghini. 2017. Multi-level simulation of Internet of Things on smart territories. *Simulation Modelling Practice and Theory* 73 (2017).
[10] 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.
[11] Alexei Ledenev et. al. [n.d.]. *Pumba: Chaos testing tool for Docker.* Retrieved 30/08/2019 from https://github.com/alexei-led/pumba
[12] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach. 2019. MockFog: Emulating Fog Computing Infrastructure in the Cloud. In *Proc. of the First IEEE International Conference on Fog Computing*.
[13] R. Hashemian, D. Krishnamurthy, and M. Arlitt. [n.d.]. Web Workload Generation Challenges - an Empirical Investigation. *Softw. Pract. Exper.* 42 ([n. d.]), 629–647. https://doi.org/10.1002/spe.1093
[14] R. Hashemian, D.r Krishnamurthy, M. Arlitt, and N. 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.
[15] M. Kovatsch, M. Lanter, and Z. Shelby. 2014. Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the*. IEEE, 1–6.
[16] K. Kuladinithi, O. Bergmann, Th. Pötsch, M. Becker, and C. Görg. 2011. Implementation of coap and its application in transport logistics. *Proc. IP+ SN, Chicago, IL, USA* (2011).
[17] P. Levis et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.
[18] Ph. Levis, N. Lee, M. Welsh, and D. Culler. 2003. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proc. of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/958491.958506
[19] V. Looga, Zh. Ou, Y. Deng, and A. Yla-Jaaski. 2012. Mammoth: A massive-scale emulation platform for internet of things. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, Vol. 3. IEEE, 1235–1239.
[20] R. Love. [n.d.]. *CPU Affinity.* Retrieved 30/08/2019 from http://www.linuxjournal.com/article/6799
[21] Li Ma, Y. Yang, Zh. Qiu, G. Xie, Y. Pan, and Sh. Liu. 2006. Towards a complete OWL ontology benchmark. In *European Semantic Web Conference*. Springer, 125–139.
[22] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran. 2017. Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *2017 IEEE Fog World Congress (FWC)*. IEEE, 1–6.
[23] 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
[24] 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.
[25] J. Mukherjee, Diwakar. Krishnamurthy, and M. Wang. 2016. Subscriber-Driven Interference Detection for Cloud-Based Web Services. *IEEE Transactions on Network and Service Management* (2016).
[26] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and Th. Voigt. 2006. Cross-level sensor network simulation with cooja. In *Local computer networks, proceedings 2006 31st IEEE conference on*. IEEE, 641–648.
[27] G. Z Papadopoulos, J. Beaudaux, A. Gallais, Th. Noel, and G. Schreiner. 2013. Adding value to WSN simulation using the IoT-LAB experimental platform. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*. IEEE, 485–490.
[28] B. Ramprasad, J. Mukherjee, and M. 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.
[29] Mark S. [n.d.]. *collectl.* Retrieved 30/08/2019 from http://collectl.sourceforge.net/Documentation.html
[30] Z. Shelby, K. Hartke, and C. Bormann. 2014. *The Constrained Application Protocol (CoAP).* RFC 7252. RFC Editor. http://www.rfc-editor.org/rfc/rfc7252.txt http:

//www.rfc-editor.org/rfc/rfc7252.txt.

[31] W. Sobel, Sh. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, Sh. Patil, A. Fox, and D. Patterson. 2008. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, Vol. 8.

[32] G. Tanganelli, C. Vallati, and E. Mingozzi. 2015. CoAPthon: easy development of CoAP-based IoT applications with Python. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 63–68.

[33] F. Van den Abeele, E. Dalipi, I. Moerman, P. Demeester, and J. 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.