TASK Quarterly 27 (1) 2023

Analysis of cores affinity within the containerized environment based on selected IOT middleware - observations and recommendations

Robert Kałaska
Faculty of Electronics
Telecommunications and Informatics
Gdansk University of Technology
Narutowicza 11/12, 80-392, Gdansk, Poland

https://doi.org/10.34808/0n9g-zh19

Abstract

The Internet of Things gets bigger and bigger audiences. This topic is really popular in science and also in industry. There are many fields for research. One of them is efficient deployment against resource utilization. Another one is containerization within IoT platforms. One of the commonalities of these two topics is different CPU affinity against containerized platforms to get the best performance. There were plenty of papers dedicated to containerization even in IoT but none of these focused on core affinity. As this survey analyzes the scalability and stability of the platform in different core-container configurations based on the IoT platform - DeviceHive, it brings a novelty to this area. Most interesting observations were made in the field of the same configurations in terms of the number of nodes but varying with core affinity. Analyzed observations may be useful during the architecture planning phase for containerized IoT platforms.

Keywords:

IoT, HPC, Containerization

1. Introduction

For the last few years, many people have focused on the Internet of Things. The latest research showed that there are around 50 billion devices linked with the IoT (end of 2020) [1] and future forecast aims at 500 billion devices connected in the IoT network [2]. There are many research papers in this field dedicated especially to cloud computing [3], processing on edge [4][5] and containerization of IoT platforms [6] [7]. Taking an in-depth look at these articles we can find out that containerization is a preferable technique to set up an isolated environment. It has a lower footprint than virtualization. One of the most commonly used containerization tools is Docker [8]. It may be interesting to see if different CPU cores affinity of the same number of cores affects the performance of the proposed platform setup and if so in what manner. This article proposes different platform configurations – varying with node numbers running middleware server - Docker nodes. The main goal of the research was to check if attaching a limited and unique set of CPU cores to each container provides better results than attaching more cores but shared along with other containers. What is important – the total CPU cores available for the whole platform stay the same on each configuration. The original paper was presented in [9] but as the results may be interesting for a wider group of researchers I also decided to present these results in English. Moreover, there are also additional figures and short analyses which were not presented in previous work.

The paper is structured as follows. Section 2 presents related work in this area. Section 3 describes different techniques for obtaining isolated environments - virtualization and containerization and their differences. Section 4 provides information on how kernel controls the CPU affinity and what kind of algorithms for core assignment are used within kernel and Docker. Section 5 evaluates the proposed test environment and metrics. Section 6 discuss the obtained results and finally Section 7 concludes this research.

2. Related work

To achieve better performance using CPU core affinity several works were already presented. In [10] authors benchmarked microservice architecture using TeaStore which is an open source microservices benchmark. They analyzed results against testbed hardware. After analysis, researchers proposed optimization using CPU core affinity which resulted in an uplift of 22 % in throughput and a latency reduction of about 18 %. Another survey [11] aimed at core affinity against file upload in Hadoop - an open-source platform for efficient processing and storing large datasets. The authors ran a prepared benchmark with different core affinities and analyzed the results. After an in-depth analysis of the obtained results, they proposed a framework for dynamic core affinity. Their solution improved 42 % throughput in the Hadoop file upload scenario. There is also a study [12] dedicated to web server (Lighttpd) scalability. The authors analyzed different workload configurations under different core affinity setups. Researchers observed scalability improvement up to 45% using the Balancing network interrupt handling load method in TCP/IP intensive workload configuration.

Many researchers also focused on performance surveys between containerized, virtual, and bare-metal environments. In [13] author proposed a macro-benchmark performance comparison between two different technologies - virtualization using Xen and containerization using LXC. Obtained results showed that performance is better on LXC but in the testbed, against the ability to isolate resources, Xen gained 200 times better results. The author comes with the observation that containerization may be better fitted for PaaS clouds while virtualization may be fitted for IaaS clouds. In-depth analysis of different containerization technologies and virtualization with Xen on HPC servers were made in [14]. Each of the benchmarked containerization solutions (LXC, OpenVZ, and VServer) represented much better results in comparison to virtualization with Xen. The provided results were also similar to bare-metal performance. Similar to [13] authors also found out that isolation of resources is not working properly in a containerized environment resulting in up to 89.6 % drop in performance when another container on the same machine is stress-loaded. Another interesting survey was made in [15]. The authors compared different containerization technologies: LXD, Docker, and LXC. Each of them was compared in terms of RAM speed (97-99%), CPU intensive code (73-95%) and IO read (74-85%), write (77-83%). The average performance of Docker was 81.7%, LXD 90.5% and LXC 86.5%.

There were also some works dedicated to the comparison of different IoT middlewares such as [16], where authors compared the stability and scalability of SiteWhere and ThingsBoard middlewares. They concluded that SiteWhere had better performance results for MQTT but worse stability and ThingsBoard had better performance in REST API. A similar topic had been surveyed in [17]. It contains benchmarks of platforms Konker, Orion+STH, SiteWhere, and InatelPlat. Tests were conducted for 10000 parallel users and different packet sizes represented as various number of parameters sent. The best performance was achieved by SiteWhere.

Finally, there is also research dedicated to containerization deployment within IoT technologies. The possibility of deploying containerized nanoservices in IoT networks was analyzed in [18]. The authors proposed optimization resulting in sizing down the container size to a few tens of MBs simultaneously achieving initialization time of the whole nanoservice in less than a minute. Another work [19] is focused on the design of a mobile cloud that is built of containers deployed on IoT devices. The proposed architecture named IoTDoc was compared against the Amazon EC2 solution. Testbed were based on Swarm Managers and Node Managers deployed on both clouds. The results of Amazon Cloud were better but the proposed architecture in some resource-oriented (DD write test) cases presented comparable performance. An interesting solution dedicated to improving scalability based on proper load distribution may be found in [20]. The authors analyzed a scenario where an IoT network is built of three layers: sensor, gateway, and cloud. In the proposed architecture each gateway was running three services - reading data from sensors, processing, and transferring to the cloud. Thanks to containerization and division to microservices authors achieved proper load distribution within the proposed cloud.

None of these works analyze core affinity against containerized environments. Novelty outcomes of this work may provide an important contribution to the field of proper deployment in HPC architectures. Moreover, as tests are executed within IoT middleware platforms and IoT application scenarios, proposed observations could be directly implemented in this area, especially in the containerized deployment of cloud middleware.

3. Containerization and virtualization

Different concepts result in obtaining the isolated environment - containerization and virtualization. They differ in architecture and consequently, performance, start time, and footprint are different from each other. In this section, I will shortly describe its concept and highlight the difference.

3.1. Virtualization

In classic virtualization the component named system hypervisor allows different guest OS to run parallel on the single bare-metal machine. System hypervisor may have different implementations e.g. being part of a root OS (solutions like KVM) or running directly on bare-metal (e.g. ESXi). Each parallel-running guest OS obtains its components for computing, storage, and network. Guest OS has its own ISA. Translation of its instructions to host ISA may be different (e.g. Hardware Virtualization or Binary Translation) [21].

3.2. Containerization

Containerization is the lightweight concept of virtualization. The main idea is that each container is running on the same host kernel instance. Behind the scenes it is implemented using the concept of cgroups and namespaces [22]. Using the above solution its possible to isolate the whole process while all resources are controlled by one host kernel.

3.3. Comparison of both technologies

Virtualization is made on the hardware layer so obviously its start time is much higher (measured in minutes) while the containerization to start needs only to spawn a new process so its start time is much lower (measured in seconds). There were many types of research dedicated to performance comparison. In [23] authors compared Docker and KVM using such tools as Sysbench, Phoronix, and Apache and showed that Docker had better results in all tests. Another research [24] provides us with the conclusion that Docker has a noteworthy better performance than KVM. These show that containerization has a much lower footprint. In [25] in-depth comparisons were made against KVM, Docker, and bare-metal. Results also showed a visible difference in favor of Docker against KVM. Moreover, the performance of the containerized environment was quite similar to bare metal. No significant performance drops were observed. On the other hand, using containerization it is not possible to run another operating system when using virtualization it is possible to run e.g. Linux in parallel with Windows. In the figure 1 both architectures are presented.

4. CPU affinity

In [26] we can find a description of CPU affinity as the ability to bind one or more processes to one or more processors. With this feature, we can completely control which process is running on which core of the CPU. Using such a solution it is possible to improve the performance of the provided platform but it needs to be done carefully because invalid setting of processes against cores may also lead to aggravate performance. Kernel realizes CPU affinity using the scheduler. It is important to know that the Linux scheduler naturally tries to keep the chosen process running as long as possible on the same cores to avoid switching between different cores. Linux and Docker are both using the same scheduler named CFS - Completely Fair Scheduler [27]. Its main idea is to simulate an ideal processor that can divide its resources between many tasks. Based on this concept its implementa-

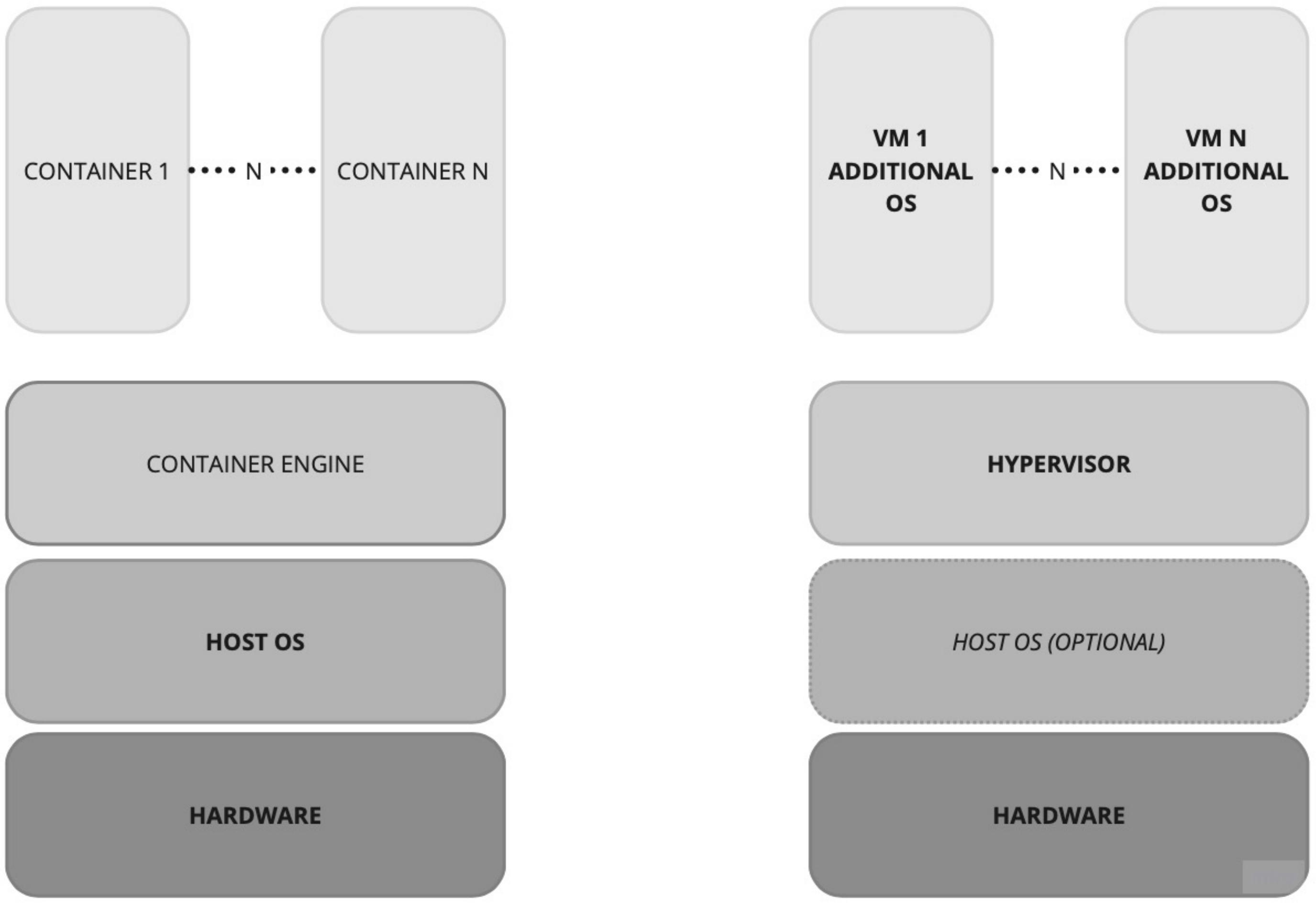

Figure 1: Architecture overview of containerization (left side) and virtualization (right side)

tion calculates how much time each process is waiting for the CPU and the one with the longest wait time is getting the resources. A detailed overview of this solution may be found in [28]. In this research, I simulated different affinity with command *taskset* [29]. It allows to binding chosen CPU to the chosen PID or started process.

5. Application model and testbed description

IoT solutions are build of 3 layers:

1. sensors - standalone microelectronic devices
2. gateways/middlewares - brokers able to analyze and process data before sending it further
3. cloud platforms - gathering data and realizing advanced business logic

The proposed application model involves the first two layers. Sensors are simulated using a cluster of computers while middleware is deployed on an HPC machine. Devices send notification messages continuously in a loop and middleware receives them and collects them in a fast runtime in-memory database. Such messages might be further processed by the cloud platform which is out of the scope of this scenario. A similar application model is proposed in [30]. The authors presented an IoT solution aimed at measuring pollution levels. Sensor reads were sent continuously to the server which processed the data. In this case, I decided to send data continuously - without any delay between each packet in favor of simulating a high load on the middleware side.

To fully discover the impact of core affinity each step beginning with architecture setup through software and hardware configuration and ending with test configuration must be done carefully. The proposed architecture focuses on stable processing of high load from end devices and thanks to the load balancer it can scale horizontally. The simulator of end devices should be stable and repeatable, such as proposed in the following configuration. The target software platform is dedicated to IoT and as this survey is focused on containerization it should be easily adaptable allowing quick startup within the containerized environment. Each test case should provide reliable and repeatable data. To achieve this goal the tests were repeated and the proposed test time provided us with hundreds of thousands of samples to analyze.

5.1. Architecture overview

In the proposed survey the architecture consists of device simulators that connect through the load balancer to the middleware platform. Depending on the tested configuration there might be different numbers of nodes running the middleware platform and each of them runs with different core affinity. Middleware platforms within duplicated nodes exposed only its microservices. All common (in terms of middleware) 3rd party software (Kafka, Redis) was running on an independent container. It allows to saving of runtime data to the same queue and cache and also sharing configuration in one Postgres database. The proposed architecture is presented in figure 2.

5.2. Device simulator

Device simulators were deployed on a cluster of 29 computers equipped with Intel Xeon CPU E5345 @2.33GHz, 8 GB RAM, and running CentOS 6. Each machine was running in parallel with several dozen client simulators - beginning with 30 devices up to 90 devices per machine. This gives a total load in the range of 870 devices to 2610 devices working in parallel. Client simulator was a Java application which constantly in a loop generated 36 random bytes of data (sensor read) and sent it using REST API with HTTPS to the middleware server. Each client had a predefined time frame of work - 10 minutes - and all of them were started together.

5.3. Middleware platform and hardware

Nowadays web applications are based on microservices. As the middleware for IoT presents a similar functionality I decided to choose a platform that is built using microservices. Moreover, its architecture should not be too fragmented to not consume much time on startup. After analysis of available solutions, I decided to choose DeviceHive. It is composed of 3 microservices:

- Frontend - to expose communication API of the services with REST API and WebSocket
- Authorization - to realize authentication and authorization of users and provide them JWT tokens to access the Frontend service
- Backend service which realizes application logic.

Moreover, platforms also use 3rd party libraries like databases (default Postgres) to store platform configuration and users. To provide a low-latency and scalable solution for data processing it runs Kafka (queue platform) and Redis to store all messages during system runtime.

The above solution was deployed on Docker containers running on a high-performance server with 2 Intel Xeon CPUs E5-2680 v2 @2.8 GHz with a total of 40 logical processors (with HT) equipped with 128 GB of RAM running Ubuntu.

5.4. Load balancer

Load balancer is running *nginx* software. It was configured to expose two servers - one for authorization and one for frontend service. Each of them was running with load balancing using round-robin algorithm. Worker connections and backlogs were set to 9000 to avoid bottleneck on load balancer. Moreover SSL termination was on load balancer and further communication was in plain HTTP. Load balancer was running on standalone physical machine equipped with 20 cores Intel Xeon CPU and 64 GB of RAM.

5.5. Testbed description

The test begins with startup of each containers and its services. After that 3 different messages are sent to "warm-up" the platform - allow load all needed classes to JVM and initialize all additional software. These messages are skipped in further analysis. When "warming-up" messages are successfully processed the main test part begins. All clients starts to sends its messages. This part takes 10 minutes. If this time is up the shutdown command is sent and data is collected from all containers and clients. Detailed analysis of collected data provide us with following information:

- quantity of processed messages,
- businesses logic processing time measured on first servlet of middleware
- correctness of processing - information on middleware side
- request-response processing time

Above testbed were repeated for following configurations of middleware deployment:

- 2 nodes with 0-9 cores for node 1 and 10-19 cores for node 2
- 2 nodes with 0-19 cores assigned for both
- 5 nodes with 0-3 cores for node 1, 4-7 cores for node 2, 8-11 cores for node 3, 12-15 cores for node 4, 16-19 cores node 5
- 5 nodes with 0-19 cores assigned for each node
- 7 nodes with 0-2 cores for node 1, 3-5 cores for node 2, 6-8 cores for node 3, 9-11 cores for node 4, 12-14 cores for node 5, 15-17 cores for node 6 and 18-20 cores for node 7
- 7 nodes with 0-20 cores assigned for each node

Core affinity was achieved using *taskset* command within the container. Tested machine was equipped with 2 CPUs,

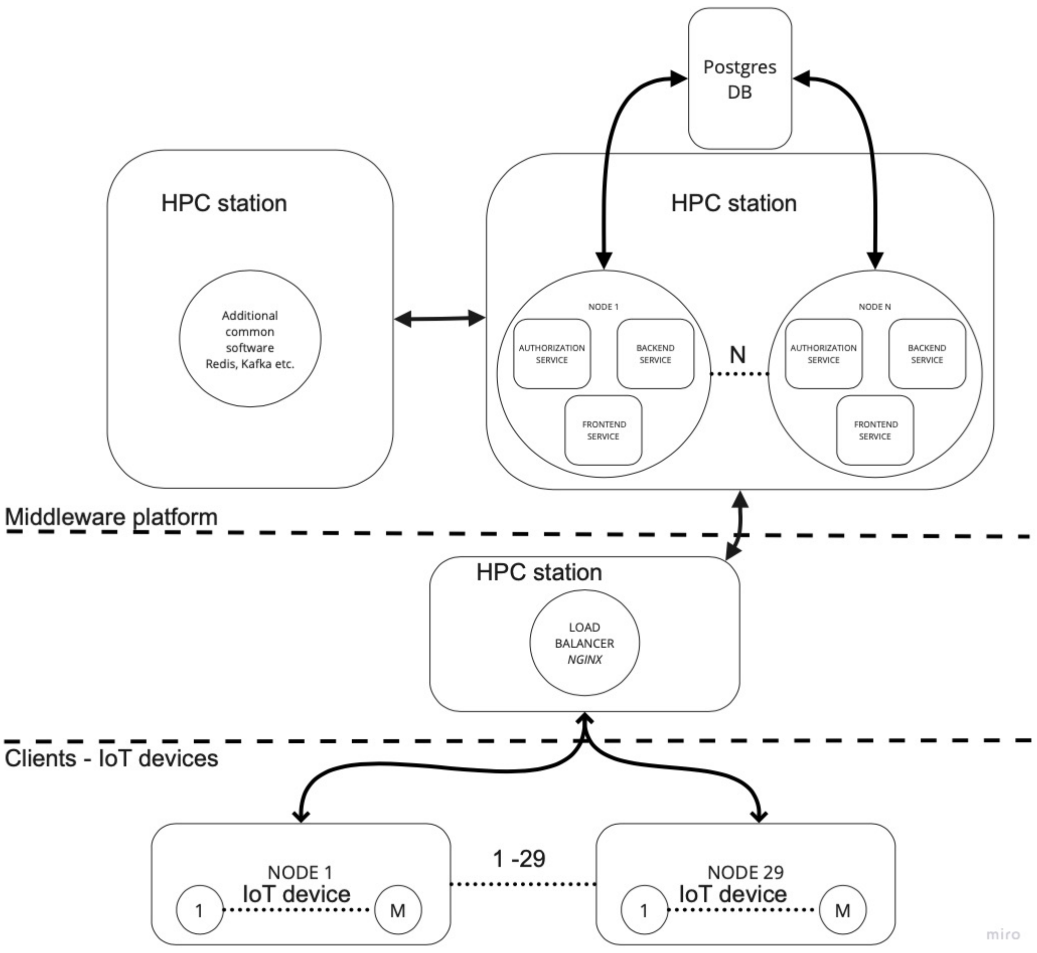

Figure 2: Architecture of tested platform

each of physical 10 cores. 0-9 cores represented physical cores of CPU 0 and 10-19 cores represented physical cores of CPU 1.

The proposed testbed was repeated 7 times. Statistical analysis of the gathered results was made. Based on the median of processing time the best and the worst test run was removed, to filter results and present smoother graphs. After that, the median of processing time, the mean value of processed message quantity, and error rates were calculated over the rest results. Such results were presented in the graphs. Moreover, to introduce the reader to the results of time processing values of the first and third quartile were presented as well as median in corresponding tables 1, 2, 3.

6. Proposed metrics

This paper is aimed at a performance comparison between one multi-node configuration where the predefined pool of cores is assigned to each node against the same configuration with the same total number of cores but divided into an equal number of cores independent for each node. To provide descriptive results four values were measured:

1. number of successfully processed requests,
2. error rate,
3. frontend service processing time,
4. client processing request-response time,

The first one represents the number of successfully processed requests along the whole test - a 10-minute time frame. Successfully means that middleware responds with proper acknowledgment for incoming messages. The second parameter presents the percent of error responses. It is calculated as the number of error responses to all responses generated by the middleware platform. The third metric shows the processing time on the server side measured on the first servlet inside the frontend microservice. It is important to say that total processing in server time also contains other parts of processing before and after frontend microservice. The last measure is the time calculated on the client side between request and response - so the processing time of service from the client's point of view.

7. Results presentation and analysis

Processing time measured on server side inside frontend service of analyzed middleware is presented in figure 3 and corresponding tables 1, 2, 3. There are two visible observations. When more cores are involved in processing the processing time is much lower (2 node configurations have much less time than others) and it works as expected. Improving the number of cores improves scalability. This observation was analyzed in-depth in article [31]. The second and important observation, which this paper aims for, is that configurations with the same node numbers but varying in core affinity have visibly different results. When individual cores are assigned to a node processing time is more stable and significantly lower. This means that JVM which realizes application logic is more efficient when cores are assigned individually per container. Corresponding to the research question – at this level, it is clearly visible that attaching cores individually to a container provides better results – stable and faster. It is important to note that this time is measured inside a servlet, so it does not show server delays, OS layer processing, and other operations required to properly resolve HTTP requests. On the opposite in the figure 4 it is visible that the total number of successfully processed messages is a bit better for configurations with more cores available but shared within many nodes. This means that total performance is better in such configurations. It shows that all software involved in processing HTTP requests, until application logic servlet, is able to process more requests when more cores are available, even if these cores are shared with other nodes. Moreover, taking an in-depth analysis of the error rate presented in 5 shows that better performance in such configuration has a significantly higher error rate. It is definitely visible on configurations with 5 and 7 nodes. The error rate for those with individual cores is around 0 % and the same node configuration with shared cores is between 2 - 4 %. Additional measures realized from a client point of view are presented in figure 6 and corresponding tables 4, 5, 6. According to the research question attaching individual cores per container provides us with better results until the medium load (middle of load on processed message quantity from figure 4), after that, it is reversed – shared cores within many containers achieves better results. However, these differences are not as significant as measured on the server side.

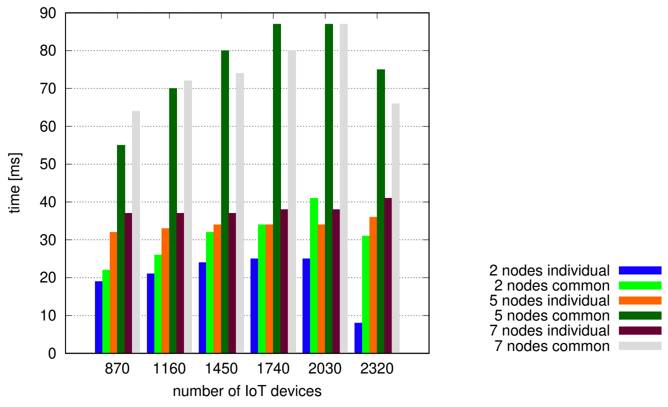

Figure 3: Processing time inside frontend service

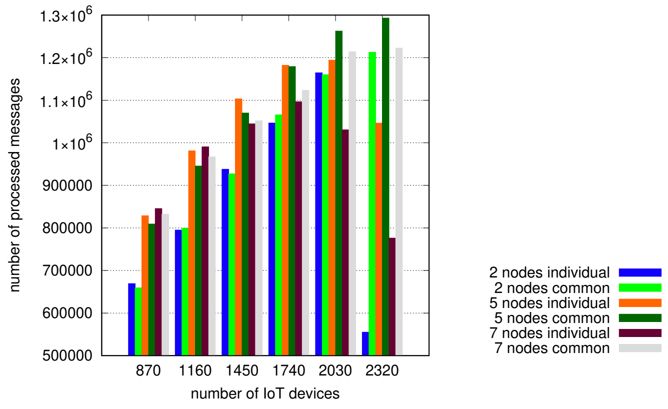

Figure 4: Number of successfully processed messages

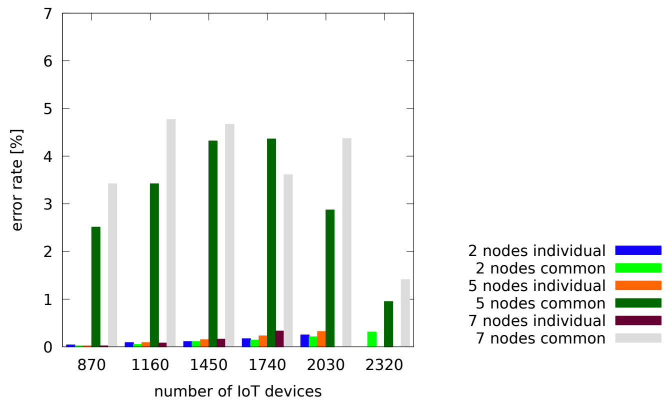

Figure 5: Errors rate

8. Conclusions and future work

Optimization to utilize hardware most efficiently is an important topic. In this survey, I analyzed if attaching a limited and unique set of CPU cores to each container provides better results than attaching more cores but shared along with other containers. Prepared real-world application examples showed that under high load configurations with shared cores presented about 10 % better performance but with the cost of a higher error rate. Configuration with unique CPU cores attached presented more reliable results with lower values of application logic pro-

Table 1: Processing time inside frontend service - 2 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870.00	9.74	10.20	19.76	22.93	36.49	47.65
1160.00	10.02	10.75	21.26	26.10	39.02	52.85
1450.00	11.95	12.82	24.47	32.30	42.92	60.37
1740.00	12.64	13.17	25.94	34.11	45.39	63.41
2030.00	11.24	15.85	25.56	41.02	48.13	75.23
2320.00	5.15	11.92	8.41	31.81	28.29	69.06

Table 2: Processing time inside frontend service - 5 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870	18.35	25.65	32.60	55.68	51.13	88.76
1160	19.36	32.65	33.75	70.52	51.77	103.45
1450	19.65	37.32	34.34	80.42	52.55	113.12
1740	19.35	39.77	34.29	87.48	53.73	122.83
2030	18.82	36.51	34.87	87.80	57.74	131.35
2320	18.09	29.96	36.17	75.49	63.29	133.02

Table 3: Processing time inside frontend service - 7 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870	22.02	31.52	37.70	64.50	57.61	91.08
1160	21.82	34.79	37.89	72.89	58.28	99.84
1450	21.43	31.16	37.75	74.64	58.20	106.79
1740	21.57	25.38	38.19	67.33	60.90	107.95
2030	20.88	39.71	38.79	88.00	63.98	133.56
2320	20.57	25.92	41.11	66.10	71.12	133.41

Table 4: Request - response processing time - 2 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870.00	200.06	215.69	308.76	328.10	755.14	789.87
1160.00	251.03	267.60	412.89	421.85	919.98	905.03
1450.00	318.84	331.24	494.55	501.31	966.19	921.92
1740.00	381.06	408.91	562.17	573.56	1006.04	933.70
2030.00	443.47	471.68	623.71	630.87	1019.60	959.28
2320.00	348.20	507.26	1088.70	682.74	4218.92	1177.86
2610.00	324.53	358.50	963.89	1275.15	6087.12	5542.19

Table 5: Request - response processing time - 5 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870.00	268.99	271.90	406.79	405.91	658.59	651.91
1160.00	343.13	351.13	482.63	484.46	696.34	695.38
1450.00	395.24	402.71	531.22	538.45	747.67	736.97
1740.00	435.91	456.35	569.97	592.24	835.20	784.52
2030.00	458.20	517.05	596.91	655.53	1110.79	844.86
2320.00	430.60	596.45	563.27	740.05	1813.22	966.98
2610.00	330.60	587.61	461.21	744.14	1273.55	1100.60

Table 6: Request - response processing time - 7 nodes individual and common

Number of IoT devices	First quartile [ms]		Median [ms]		Third quartile [ms]	
870.00	338.70	308.48	456.59	418.31	621.10	588.17
1160.00	371.64	372.83	485.73	484.24	660.98	640.01
1450.00	438.90	424.37	571.59	547.35	811.08	714.49
1740.00	447.01	462.39	563.46	598.87	821.27	786.27
2030.00	442.44	551.08	584.70	680.13	961.54	859.03
2320.00	403.14	585.69	536.72	747.12	1341.09	1025.49
2610.00	361.59	639.76	494.26	831.77	1391.27	1246.60

cessing time. Taking into account the provided observation it could be stated that up to the proposed application scenario, both assumptions are correct. When high reliability is expected it is better to define core affinity for

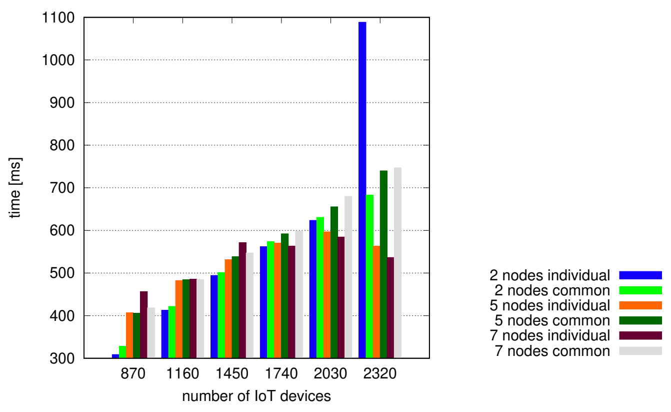

Figure 6: Request - response processing time

individual containers during the deployment phase, but when crucial is performance it is better to provide containers with more CPU cores but shared within a cluster of containers. Considering the real-world scenario, where applications are deployed in the cloud managed by data centers it is important information to the client if provided resources like CPU cores are assigned to the individual application or may be shared within many products. IT professionals based on such information and results presented in this paper may choose the best deployment scheme that fits their needs. Considering future work in-depth investigation should be done at the server and OS layer to determine if tuning this field may provide better results in such a scenario.

References

[1] R. G. S. Rethinavalli, "Botnet attack detection in internet of things using optimization techniques," *International Journal of Electrical Engineering and Technology (IJEET)*, vol. 11, no. 4, 2020.

[2] Y. B. Zikria, R. Ali, M. K. Afzal, and S. W. Kim, "Next-generation internet of things (iot): Opportunities, challenges, and solutions," *Sensors*, vol. 21, no. 4, 2021.

[3] H.-L. Truong and S. Dustdar, "Principles for engineering iot cloud systems," *IEEE Cloud Computing*, vol. 2, no. 2, pp. 68–76, 2015.

[4] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating performance of containerized iot services for clustered devices at the network edge," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 1019–1030, 2017.

[5] C. Savaglio, P. Gerace, G. Di Fatta, and G. Fortino, "Data mining at the iot edge," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–6, 2019.

[6] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of microservices for iot using docker and edge computing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118–123, 2018.

[7] M. S. Abdul, S. M. Sam, Norliza, Mohamed, K. Kamardin, R. Akmam, and Dziyauddin, "Docker containers usage in the internet of things: A survey," 2019.

[8] V. G. da Silva, M. Kirikova, and G. Alksnis, "Containers for virtualization: An overview," *Applied Computer Systems*, vol. 23, pp. 21–27, May 2018.

[9] M. Affek, S. Barański, T. Boiński, P. Gładkowska, T. Gruzdzis, W. Janowski, R. Kałaska, H. Krawczyk, J. Kuchta, R. Lipiński, J. Łuczkiewicz, M. Matuszek, S. Olewniczak, P. Orzechowski, K. Selwon, A. Szamocki, J. Szymański, A. Wawrzyński, K. Wicki, and K. Zawora, "Algorytmy i zastosowania inteligencji obliczeniowej (kaskbook 2022)." online.

[10] S. Caculo, K. Lahiri, and S. Kalambur, "Characterizing the scale-up performance of microservices using teastore," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 48–59, 2020.

[11] J.-Y. Cho, H.-W. Jin, M. Lee, and K. Schwan, "On the core affinity and file upload performance of hadoop," in *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, DISCS-2013, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2013.

[12] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson, "Improving the scalability of a multi-core web server," ICPE '13, (New York, NY, USA), p. 161–172, Association for Computing Machinery, 2013.

[13] T. Scheepers, "Virtualization and containerization of application infrastructure: A comparison."

[14] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240, 2013.

[15] A. R. Putri, R. Munadi, and R. M. Negara, "Performance analysis of multi services on container docker, lxc, and lxd," *Bulletin of Electrical Engineering and Informatics*, vol. 9, pp. 2008–2016, October 2020. 10.11591/eei.v9i4.1953.

[16] A. A. Ismail, H. S. Hamza, and A. M. Kotb, "Performance evaluation of open source iot platforms," in *2018 IEEE Global Conference on Internet of Things (GCIoT)*, pp. 1–5, 2018.

[17] M. A. da Cruz, J. J. Rodrigues, A. K. Sangaiah, J. Al-Muhtadi, and V. Korotaev, "Performance evaluation of iot middleware," *Journal of Network and Computer Applications*, vol. 109, pp. 53–65, 2018.

[18] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila, "Docker enabled virtualized nanoservices for local iot edge networks," in *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, pp. 1–7, 2019.

[19] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, *IoTDoc: A Docker-Container Based Architecture of IoT-Enabled Cloud System*, pp. 51–68. Cham: Springer International Publishing, 2020.

[20] B. Ahmed, B. Seghir, M. Al-Osta, and G. Abdelouahed, "Container based resource management for data processing on iot gateways," *Procedia Computer Science*, vol. 155, pp. 234–241, 2019. The 16th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2019),The 14th International Conference on Future Networks and Communications (FNC-2019),The 9th International Conference on Sustainable Energy Information Technology.

[21] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*, pp. 610–614, 2014.

[22] R. Rosen, "Namespaces and cgroups – the basis of linux containers."

[23] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[24] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers,"

in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172, 2015.

[25] S. Mazaheri, Y. Chen, E. Hojati, and A. Sill, “Cloud benchmarking in bare-metal, virtualized, and containerized execution environments,” in *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pp. 371–376, 2016.

[26] R. Love, “Cpu affinity.”

[27] K. Lynch, “Understanding linux container scheduling.”

[28] C. S. Pabla, “Completely fair scheduler.”

[29] R. M. Love, “taskset(1) — linux manual page.”

[30] C. Xiaojun, L. Xianpeng, and X. Peng, “Iot-based air pollution monitoring and forecasting system,” in *2015 International Conference on Computer and Computational Sciences (ICCCS)*, pp. 257–260, 2015.

[31] R. Kałaska and P. Czarnul, “Investigation of performance and configuration of a selected iot system - middleware deployment benchmarking and recommendations,” *Applied Sciences*, vol. 12, no. 10, 2022.

[32] K. A. O. B. M. W. H. Adhitya Bhawiyuga, Dany Primanita Kartikasari, “Architectural design of iot-cloud computingintegration platform,” *TELKOMNIKA*, vol. 17, pp. 1399–1408, June 2019.

[33] F. Samie, V. Tsoutsouras, L. Bauer, S. Xydis, D. Soudris, and J. Henkel, “Computation offloading and resource allocation for low-power iot edge devices,” in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 7–12, 2016.