NimbleNet: Serverless Computing for the Extreme Edge in Factory Environments

Kilian Müller*
kilian.felix.mueller@fau.de
Institute for Smart Electronics and Systems,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Erlangen, Germany

Maximilian Seidler*
maximilian.seidler@fau.de
Department of Computer Science 4,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Erlangen, Germany

Peter Ulbrich
peter.ulbrich@tu-dortmund.de
Department of Computer Science 12,
Technische Universität Dortmund
Dortmund, Germany

Norman Franchi
norman.franchi@fau.de
Institute for Smart Electronics and Systems,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Erlangen, Germany

Abstract

Modern factories are moving towards modular layouts with more generic production cells, allowing rapid production customization. Here, the cell's edge nodes exhibit considerable heterogeneity, ranging from simple microcontrollers to fully-fledged computing systems, which mandates that assembly-specific serverless functions used by Automated Guided Vehicles (AGVs) navigating through and linking the production cells are platform-independent. These moving manufacturing processes call for an adapted communication and distribution infrastructure to maintain low response times while minimizing network traffic.

We present NimbleNet, a lightweight distribution and orchestration approach for serverless functions, specifically tailored for IoT devices in industrial factory settings. NimbleNet leverages lightweight sandboxing such as WebAssembly, dynamic dependency management, and a neighbor-first caching strategy to enable efficient, platform-independent deployment of tasks at the extreme edge. Our simulations and real-world evaluations demonstrate that our approach facilitates the dynamic deployment and execution of tasks on resource-constrained devices while optimizing the network load within a mesh network configuration.

*Both authors are affiliated with Siemens AG, Research and Predevelopment. Both authors contributed equally to this research.

WOSC '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-1336-1/24/12
https://doi.org/10.1145/3702634.3702953

CCS Concepts: • **Software and its engineering** → **Distributed systems organizing principles**; • **Computer systems organization** → *Embedded and cyber-physical systems*; • **Networks** → *Mesh networks.*

Keywords: Serverless computing, Edge computing, IoT, WebAssembly, Dynamic task deployment, Network efficiency

ACM Reference Format:
Kilian Müller, Maximilian Seidler, Peter Ulbrich, and Norman Franchi. 2024. NimbleNet: Serverless Computing for the Extreme Edge in Factory Environments. In *10th International Workshop on Serverless Computing (WOSC '24), December 2–6, 2024, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702634.3702953

1 Introduction

The increasing complexity and customization of products, coupled with the unpredictability of global markets, has necessitated the development of more adaptable manufacturing processes. These are tailored to smaller batch sizes and designed to be flexible and responsive to changing market demands. Hence, companies are transitioning from line production, which is limited by factory layout, to cellular layouts. This shift is a strategic move towards a more adaptable approach, resulting in a process-driven factory where the manufacturing is specifically tailored to each workpiece.

Without traditional fixed production stations and conveyor systems, Autonomous Guided Vehicles (AGVs) have emerged as the new backbone of the manufacturing process. They primarily transport workpieces but also facilitate production by carrying components and information about the workpiece states, precedent, and ascendant processing steps. In this paper, we urge the AGV not only to be equipped with the tangible elements necessary for the manufacturing process, but also to describe the manufacturing steps at each cell. Thus, the AGV guides the assembly sequence by coordinating the machines and managing their interactions. This software integration consolidates manufacturing

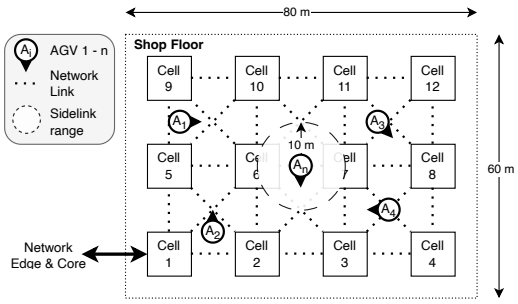

Figure 1. Factory floor with manufacturing cells each hosting IoT nodes, connected via a mesh network. AGVs navigate the shop floor and can connect to nodes within a 10 m radius.

resources within the AGV and enables customized process configurations and models with minimal effects on network traffic, addressing congestion issues frequently encountered in industrial wireless networks.

Motivating Example. The railway manufacturing environment, a representative case, is remarkably adaptable. Whether the target product is a wagon or a locomotive, the production process is customized, with batch sizes ranging from one to a few hundred units. This adaptability allows for the creation of custom components that meet the technical, environmental, cultural, and social demands of the operational region. Components are manufactured in production islands with task-specific tools such as assembly, welding, gluing, or lacquering, defining the cell's general capabilities. These are controlled and monitored by (real-time) Internet of Things (IoT) nodes, which offer external insights into the workpiece and the ongoing production process, facilitating the evaluation of manufacturing performance, quality, and success rate. A mesh network topology connects all nodes. Data from the IoT nodes can enhance the islands' capabilities, and combining various nodes may further augment this advantage. The overall setup is in Figure 1.

1.1 Problem Statement

Integrating the physical and logical resources for manufacturing into the AGV necessitates its interaction with the production cells. Thus, this paper addresses the issue of executing serverless functions in an industrial factory setting. We identified the following primary challenges:

Challenge 1: Heterogeneous & constrained computing platforms. Hardware platforms in shop floors are inherently heterogeneous, from IoT nodes built on price-sensitive microcontrollers with limited memory to high-power Programmable Logic Controllers (PLC). Consequently, integrating state-of-the-art serverless and orchestration software is not possible. ***Our approach:*** We leverage the WebAssembly (WASM) bytecode format for platform independence. We modified the WebAssembly Micro Runtime (WAMR) for tiny devices and present a novel approach to stateful intermittency and subsequent continuation of WASM programs.

Challenge 2: Communication overhead & uneven distribution. Wireless communication bandwidth is becoming scarce, for example, due to Machine Learning (ML) classifiers boosting traffic or devoting real-time transmission channels in Time Sensitive Networks (TSN). Consequently, efficient load balancing that leverages domain knowledge of the physical environment is essential to prevent congestion from impacting or paralyzing the production process.
Our approach: The use of efficient nearest-neighbor caching and the knowledge derivable from the manufacturing process allows us to exploit local dependency, thereby reducing and evening the network load. Furthermore, using transparent partial updates alleviates the burden on the network.

1.2 Contribution and Outline

To address the challenges mentioned above, we propose NimbleNet, a novel approach for efficient serverless computing in the extreme edge. We claim the following contributions:

1. We demonstrate how the relationship between manufacturing processes can be leveraged to split programs into smaller subroutines, enabling efficient local and nearest-neighbor caching and load balancing.
2. We present a novel approach to the intermittency and continuation of WASM program execution. The approach involves stringing together WASM states and leveraging the error mechanism to escape execution without function termination.
3. We introduce our NimbleNet prototype, designed for resource-constrained devices. Furthermore, we provide extensive measurements on a real-world test bench and simulation results to affirm our approach.

The remainder is organized as follows: In Section 2, we present potential solutions from the literature and discuss why they are not sufficient. Section 3 and 4 outline the inner workings of NimbleNet. Section 5 describes the evaluation scenario, and Section 6 presents the results. We conclude with a summary in Section 7.

2 Related Work

Before outlining our methodology, we want to highlight existing research in related fields to demonstrate that although the approaches may appear similar, they do not address the fundamental issues. The transfer of workloads across devices, particularly from IoT nodes to edge or cloud services, is a topic that has been extensively researched. Given that WASM already addresses some of the aforementioned challenges, a plethora of works also exist within the context of cloud migration and offloading. Microservice architecture [3] allows for applications being composed of small, independent services that communicate over well-defined

APIs. Well-established frameworks are Kubernetes for packaging and orchestrating microservices, Eureka for discovery, NGINX for routing, and RabbitMQ for messaging. Nurul-Hoque and Harras [10], Kreutzer et al. [6], and Nieke et al. [9] have developed frameworks facilitating strong and weak migration, which involve transmitting or omitting the application's state, respectively. Li et al. [7] advocate for the offloading of discrete functions to the edge, aided by static source code annotations. Ouacha [11] enhances the OLSR protocol for VM transmission. Cloud4IoT [12] provides an execution format-agnostic solution employing Kubernetes agents and OpenStack middleware, akin to Benomar et al. [4]. Ada-Things [13] provides load-balancing strategies for VM migration, adapting the migration method by analyzing memory page modifications. Nevertheless, all these solutions are tailored for environments like cloud, edge cloud, network infrastructure devices, or fog, with the assumption of sufficient resources, i.e., tens of MBs of RAM and generally a Linux OS, thus unsuitable for numerous IoT nodes with limited capabilities. Multi-node scheduling is not tackled in none of these studies. Yousafzai et al. [15] propose a process-based migration needing a compatible process model, specifically Linux and similar architectures. Wu et al. [14] include proximate IoT nodes as offloading targets. Their prototype uses HQEMU [5], a retargetable dynamic binary translation based on QEMU and LLVM, which, along with expensive memory remapping, requires powerful edge devices. Dynamic linking methods like LLL [8], Zephyr's LLEXT [2], and Contiki's dynamic loader [1] are complex, error-prone, and highly architecture and OS specific.

3 Approach

As detailed in Section 1, we consider an AGV, a key element in our manufacturing process, navigating through the shop floor. It efficiently transports both physical and digital assets, playing a crucial role in component assembly. Upon reaching an assembly island, it directs the tools to execute the required manufacturing steps, thereby invoking the execution of serverless functions on them.

To reduce the strain on the wireless network, it is essential to employ adapted protocols that facilitate efficient network traversal and minimize the overall transmission size. This can be achieved by increasing information density using techniques such as data compression or excluding redundant data, which can be obtained from alternative sources, i.e., caching. NimbleNet focuses on the latter. Given the constraints of the factory shop floor environment, we urge utilizing process and environmental insights to enhance caching efficiency.

Analyzing control applications for factory islands, we found that algorithms can be divided into three primary domains. The *hardware-defined* domain involves tasks specific to the configuration of devices, including sensors and actuators. These are tightly coupled to the device architecture and its peripherals. The *data-defined* domain is determined by the nature of the collected, created, or evaluated data. It is derived from different physical aspects and dimensions, such as time series analysis applicable to a range of sensors, image processing, or trajectory calculation. These are primarily mathematical algorithms used for different purposes on different devices. The *process-defined* domain pertains to workpiece-specific details, often including physical modeling of the workpieces or their quality classification.

Within the constrained setting of the industrial shop floor environment, the domain subset in use exhibits significant spatial dependency. Since hardware-defined algorithms are linked to device hardware, they are also associated with each device's spatial position. Thus, serverless functions utilizing these algorithms must run on the specific device, encouraging local caching on the device. Data-defined algorithms are intrinsically linked to the physical data. This connection arises from the interplay among the workpiece, the process itself, and the capabilities of the cell, with the latter two being interdependent. Consequently, the properties are associated with the cell and should be distributed across its devices. For example, gas detection in chemical processes typically relies on variations in electrical properties independent of the specific sensor. Data evaluation can be shared, as multiple gas detection sensors are often necessary. Similarly, the process-cell combination determines whether quality control relies on methods like electromagnetic wave analysis, vibration spectral analysis, or image-based techniques utilizing fundamental matrix or vector operations or image recognition. Consequently, caching is beneficial on the cell level. Ultimately, the process-defined functions result from the sequence of steps on the workpiece. Consequently, they are connected to the workpiece and thus (temporarily) to the AGV. As manufacturing must be as efficient as possible, critical production processes, including the AGV's path through assembly cells, are assumed to be optimized. Consequently, consecutive steps are likely to be performed in cells located closely together on the shop floor, which motivates caching in cell clusters.

With NimbleNet, we propose to use this spatial dependency and the resulting caching opportunities. As workloads are inherently unknown, a static distribution is not feasible. We utilize a heuristic approach for distributing the serverless functions at runtime among the production cells and shop floor nodes, depending on their domain. This approach enables us to achieve near-optimal caching performance by leveraging domain-specific separation. This caching behavior is particularly advantageous in manufacturing processes with small batch-to-batch or entity-to-entity variances, as demonstrated in the railway manufacturing example in Section 1.

Rather than relying on one monolithic function, NimbleNet's compilation toolchain enables compiling programs into several serverless functions that depend on each other. On execution, each call to one of the dependencies is converted into a control transfer to the NimbleNet runtime component. NimbleNet then automatically resolves this dependency and loads and executes it, including parameter and return value marshaling. These serverless functions can be derived from the program code of the manufacturing process programs by sorting and aggregating the use of functions from libraries on which the programs are built. As the number of functions is limited, our approach is based on static function maps, which enable the separation of functions into domains such as controls, time series data assessment, state-changing data assessment, and others. These can then be connected to the presented caching domains.

4 Implementation

In the following, we detail the runtime component of NimbleNet, which is a prerequisite for all nodes contributing to the system. It comprises a local cache, a scheduler, and multiple executors. The nodes are arranged in a mesh network, with nodes in close spatial proximity forming connections.

Acquisition. Upon reaching a new manufacturing cell, the AGV initiates manufacturing process execution on different nodes based on the production process by instructing the invocation of specific serverless functions. These nodes check if the requested function exists in the local cache. If found, it is labeled as *reserved*, indicating its necessity for future execution and ensuring it is not evicted. If absent, the function is retrieved from other nodes or a central registry. Acquisition is initiated with a broadcast request to direct neighbors (TTL = 1). If no neighbor responds positively, a second global broadcast is initiated. The node with the lowest response time is intrinsically chosen as the *winner*. Before starting the transmission of the binary, metadata, including the function's size, is exchanged. The local cache is assessed to verify whether it can hold the function's binary. If storage is inadequate, unnecessary functions are removed. The binary is then transferred, cached locally, and marked as *reserved*.

Execution. After acquiring a function, it may be executed using an appropriate of the executor instances. Various executors are supported when they adhere to the required API. This includes a "native" executor, which is capable of running functions tailored to specific hardware or interact with the system, if necessary. The executors are based on cooperative scheduling. When a serverless function calls upon a dependency, control is transferred to the NimbleNet runtime. Subsequently, it preserves the state, marshalls arguments, and initiates the dependency function's execution on its device, restarting acquisition if necessary. In our implementation, we developed executors for the WASM binary format and precompiled Python binaries[1]. The WAMR carries out the WASM execution. Dependency execution utilizes WASM's *import* feature via a wrapper function. This wrapper function preserves the execution state. Given the limited resources of certain IoT devices used, only the state of the serverless function is retained, enabling the reuse of the same executor instance. The state comprehensively includes the WASM call stack, operand stack, heap, and global variables. Selected runtime data is also stored to accelerate program restart. Subsequently, the wrapper function indicates an error state to the WAMR executor, immediately terminating the WASM execution. The dependency function is then loaded into the same executor instance. To restore to the original serverless function, the saved WASM state is reloaded. The call stack is then adjusted by eliminating the final stack frame and decrementing the instruction pointer, enabling the dependency call to recommence which directly evaluates to the results from the dependency.

Eviction. Once the serverless function has completed, its *reservation flag* is removed. Subsequently, the eviction approach, dependent on the system environment, takes over. For our evaluation, we employ the least recently used (LRU) strategy due to its simplicity, effectively illustrating NimbleNet's applicability.

5 Experimental Setup

The evaluation setup is aligned with the motivating example discussed in Section 1. A shop floor spans an area 80 m by 60 m, with IoT sensor nodes placed 20 m apart. When an AGV enters a 10 m perimeter around a node, it establishes a connection and periodically executes different serverless sensing and perception functions on that node to assess workpiece and overall production cell state. The simulation then encompasses network traffic and node execution.

We tested our approach on two real-life testbeds, (1) centered around thirteen edge nodes where each is a Rasbperry Pi Zero 2 W and (2) a highly-constrained testbed of twelve constrained nodes each represented by a Raspberry Pi Pico W. Former devices feature a 1 GHz BCM2835 with 512 MB RAM. Latter devices are constrained to two ARM Cortex M0+ cores operating at 133 MHz and 256 kB RAM. The evaluation presented here was performed on the highly constrained testbed (2) with limited function size to show the applicability of our approach to even the most constrained devices. However, we found the same and even elevated benefits of our approach compared to the monolithic function approach when performing the same evaluation on our *high-power* testbed (1) and scheduling larger functions like complex on-device image processing and classification for quality assurance.

[1]Python binaries serve as a format for viability assessment and lack cross-platform execution capabilities.

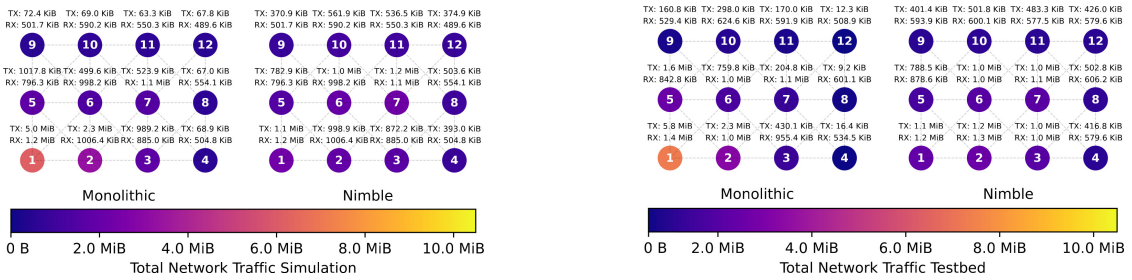

Figure 2. Network load distribution: Simulation and testbed results for monolithic vs. nimble approach

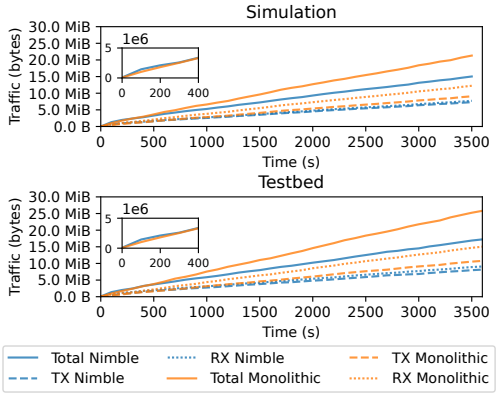

Figure 3. Total network load: Simulation and testbed results

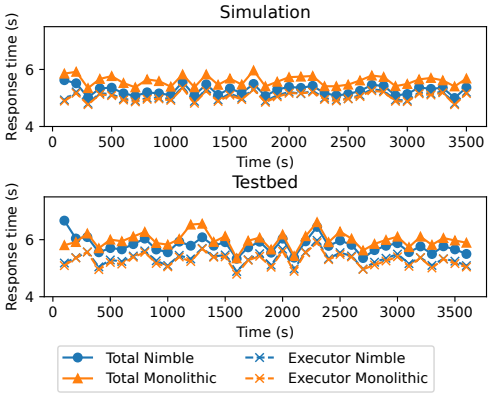

Figure 4. Response time: Simulation and testbed results

Table 1. Evaluation scenario setup parameters

Parameter	Min	Max	Parameter	Min	Max
Scenario duration (s)	3600		Cache size (bytes)	32768	
# unique MPs	200		MP response time (s)	0.1	0.5
MP memory (bytes)	100	500	# unique SFs	50	
SF response time (s)	0.01	2	SF memory (bytes)	100	500
# SF per MP	3	7	# AGVs	20	
# MPs per AGV	20		Trigger interval (s)	9	11
AGV speed (m/s)	0	1			

*MP: manufacturing process step, SF: serverless function

6 Evaluation and Results

This section assesses NimbleNet through simulation and a testbed. Six simulations are conducted with parameters specified in Table 1. These procedures are then replicated on the real-world testbed. We compare NimbleNet's dependent function sets to traditional monolithic serverless execution.

Network Traffic Distribution. Figure 2 illustrates the combined network traffic, encompassing sent (TX) and received (RX) bytes, for each node executing both NimbleNet and monolithic applications over one hour. Results are shown from our simulations on the left side, while the right side depicts the real-world testbed. Measurements reveal a four-fold and fivefold reduction in transmitted bytes for nodes closest to the network edge with NimbleNet compared to monolithic methods for simulation and testbed, respectively. The network traffic is more evenly spread across nodes, highlighting source dispersion when querying programs. Data reception remains the same across both scenarios due to identical programs being used.

Total Network Load. Figure 3 illustrates the temporal evolution of the network load as captured from the testbed and simulation. Initially, the total load for the NimbleNet method is moderately elevated, attributed to the additional overhead of routing and querying. The equilibrium is achieved at $t = 300$ s in the simulation and $t = 400$ s in the testbed. Beyond this point, the efficiency of on-device caches and nearest-neighbor caching strategies becomes apparent. Furthermore, the trend observed in simulations is evident in the practical deployment. However, the testbed shows a small rise in total network traffic relative to the simulation due to additional housekeeping packets and packet loss arising from hardware limitations in the real world.

Response Time. Figure 4 illustrates the average response time for all completed serverless functions within a 100 s sliding window. Despite the raw execution times of functions being almost identical, the NimbleNet method achieves quicker acquisition compared to the monolithic method as the caching shows its strengths. This is particularly advantageous given a substantial hop delay of ≈ 100 ms per hop in the mesh setup or when the link speed imposes a constraint, as seen with larger serverless functions. Additionally,

the slight average execution delay of NimbleNet, approximately 30 ms, arises from performing more code loadings and setups. Caching benefits are validated by previous measurements showing the same break-even point between 300 and 400 s in our real-life testbed. Figure 4 illustrates that both the simulation and the actual execution on the testbed demonstrate a similar initial pattern, where the overall response times increase significantly until most of the tasks are propagated into the system. After approximately 400 s for the testbed, the response time improvements resulting from nearest neighbor acquisition and local caching become evident, and NimbleNet outperforms the monolithic approach.

7 Conclusion

We introduced NimbleNet, a lightweight, platform-independent distribution and orchestration approach for IoT devices in dynamic industrial settings. Leveraging WebAssembly sandboxing, dynamic dependency management, and a neighbor-first caching strategy, NimbleNet enables efficient edge deployment of serverless functions on all device classes. Our simulations and IoT factory testbed evaluations demonstrate that NimbleNet reduces and balances network load across IoT nodes in a mesh network. The communication load on the sink node – the entry point to the mesh – is significantly reduced due to caching serverless functions between nodes. This caching allows nodes to share functions directly without always accessing the network edge where the functions are originally stored. Additionally, dependent serverless functions improve code reuse and optimize network efficiency. As a result, network usage decreases, and response times are faster compared to the monolithic approach. These findings indicate that NimbleNet effectively addresses the challenges of dynamic task deployment and management in heterogeneous industrial environments. By enhancing network efficiency and response times through effective caching and hierarchical code organization, NimbleNet proves well-suited for modern, modular factories that rely on dynamic task assignments and diverse hardware platforms.

Acknowledgments

This work was funded by the German Federal Ministry of Education and Research (BMBF) jointly under grant number 16KISK098 (6G-ANNA) and 16ME0454 (EMDRIVE). Special thanks to the CERN openlab for evaluating NimbleNet.

References

[1] [n. d.]. ContikiOS: The Dynamic Loader. https://github.com/contiki-os/contiki/wiki/The-dynamic-loader.

[2] [n. d.]. Zephyr Linkable Loadable Extensions (LLEXT). https://docs.zephyrproject.org/latest/services/llext/index.html.

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51. https://doi.org/10.1109/SOCA.2016.15

[4] Zakaria Benomar, Francesco Longo, Giovanni Merlino, and Antonio Puliafito. 2020. Cloud-Based Enabling Mechanisms for Container Deployment and Migration at the Network Edge. *ACM Transactions on Internet Technology* 20, 3 (June 2020), 25:1–25:28. https://doi.org/10.1145/3380955

[5] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, New York, NY, USA, 104–113. https://doi.org/10.1145/2259016.2259030

[6] Marius Kreutzer, Maximilian Leonhard Seidler, Konstantin Dudzik, Victor Pazmino Betancourt, and Jürgen Becker. 2024. Migration of Isolated Application Across Heterogeneous Edge Systems. In *2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC)*. Philadelphia, USA, 1–7.

[7] Borui Li, Wei Dong, and Yi Gao. 2021. WiProg: A WebAssembly-based Approach to Integrated IoT Programming. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488424

[8] Joy Mukherjee and Srinidhi Varadarajan. 2005. Develop Once Deploy Anywhere Achieving Adaptivity with a Runtime Linker/Loader Framework. In *Proceedings of the 4th Workshop on Reflective and Adaptive Middleware Systems (ARM '05)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1101516.1101517

[9] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3434770.3459731

[10] Mohammed Nurul-Hoque and Khaled A. Harras. 2021. Nomad: Cross-platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 168–178. https://doi.org/10.1109/IC2E52221.2021.00032

[11] Ali Ouacha. 2021. Virtual Machine Migration in IoT Based Predicted Available Bandwidth and Lifetime of Links. *International Journal of Computing and Digital Systems* 10 (April 2021). https://doi.org/10.12785/ijcds/110104

[12] Daniele Pizzolli, Giuseppe Cossu, Daniele Santoro, Luca Capra, Corentin Dupont, Dukas Charalampos, Francesco De Pellegrini, Fabio Antonelli, and Silvio Cretti. 2016. Cloud4IoT: A Heterogeneous, Distributed and Autonomic Cloud Platform for the IoT. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 476–479. https://doi.org/10.1109/CloudCom.2016.0082

[13] Zhong Wang, Daniel Sun, Guangtao Xue, Shiyou Qian, Guoqiang Li, and Minglu Li. 2019. Ada-Things: An Adaptive Virtual Machine Monitoring and Migration Strategy for Internet of Things Applications. *J. Parallel and Distrib. Comput.* 132 (Oct. 2019), 164–176. https://doi.org/10.1016/j.jpdc.2018.06.009

[14] Chao Wu, Yaoxue Zhang, and Yongheng Deng. 2019. Toward Fast and Distributed Computation Migration System for Edge Computing in IoT. *IEEE Internet of Things Journal* 6, 6 (Dec. 2019), 10041–10052. https://doi.org/10.1109/JIOT.2019.2935120

[15] Abdullah Yousafzai, Ibrar Yaqoob, Muhammad Imran, Abdullah Gani, and Rafidah Md Noor. 2020. Process Migration-Based Computational Offloading Framework for IoT-Supported Mobile Edge/Cloud Computing. *IEEE Internet of Things Journal* 7, 5 (May 2020), 4171–4182. https://doi.org/10.1109/JIOT.2019.2943176

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009