NimbleNet: Efficient Micro-Container Distribution and Orchestration for the Extreme Edge

Kilian Müller[*]
escs.mueller@fau.de
Chair of Electrical Smart City Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

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

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

Norman Franchi
norman.franchi@fau.de
Chair of Electrical Smart City Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract

The increasing complexity and customization of products, coupled with volatile markets, has motivated a shift from line production layouts to cellular production layouts, where autonomous robots are used not only for transportation but also to assist in the actual assembly of products. The problem, however, is that these general-purpose robots are inherently limited in their capabilities due to the non-process specific design. Therefore, we propose to foster collaboration with IoT nodes already present on the factory floor, introduced by the advent of Industry 4.0, which provide detailed insights and process-tailored hardware. Robots need to be enabled to trigger custom workloads on the IoT devices to gain knowledge or trigger process-related events. The challenge, however, is that integrated IoT nodes are highly constrained devices with a variety of architectures. Spatial isolation must be maintained together with a small footprint on wireless communication to prevent inference while keeping response time low, requiring efficient distribution and orchestration strategies.

In this paper, we present NimbleNet, a lightweight and platform-independent distribution and orchestration approach specifically tailored for IoT devices in the industrial setting. NimbleNet combines lightweight sandboxing approaches such as WebAssembly, dynamic dependency management, and neighbor-first caching with centralized fallback to enable efficient platform-independent deployment of tasks at the edge. We have implemented NimbleNet and evaluated the approach through simulation and on our IoT testbed demonstrating that the developed approach enables the dynamic deployment and execution of tasks on constrained devices, while concurrently reducing and balancing the network load across participating IoT nodes in our mesh configuration.

[*]Both authors contributed equally to this research.

CCS Concepts: • **Computer systems organization** → *Sensor networks*; *Sensors and actuators*; • **Networks** → Overlay and other logical network structures; • **Applied computing** → *Command and control.*

Keywords: Hierarchical Microcontainers, Edge Computing, Internet of Things (IoT), Dynamic Task Orchestration, WebAssembly (Wasm), Distributed Systems

1 Introduction

The increasing complexity and customization of products, coupled with the unpredictability of global markets, has necessitated the development of more sophisticated product manufacturing processes. As a result, many companies have moved from traditional line production to cellular layouts that allow the production process to quickly adapt to changing conditions or objectives. In the absence of fixed production positions, autonomous robots have emerged as the new backbone of the manufacturing process, not only for transporting parts but also for assisting in the actual production process.

Conversely, the integration of intelligent technologies on the factory floor is not a new phenomenon, especially with the advent of Industry 4.0. This integration has led to the emergence of highly process-tailored sensor systems, which are already being placed in optimized locations within the manufacturing station to provide access to detailed process data. Many of them contribute significantly to improving the manufacturing process by providing accurate data to visualize the process and enable the great potential of Manufacturing Execution Systems (MESs). However, the contrasting design philosophies of process-specific sensors and general-purpose actors in the form of robots hinder the convergence of these two domains, making the collaborative process ineffective.

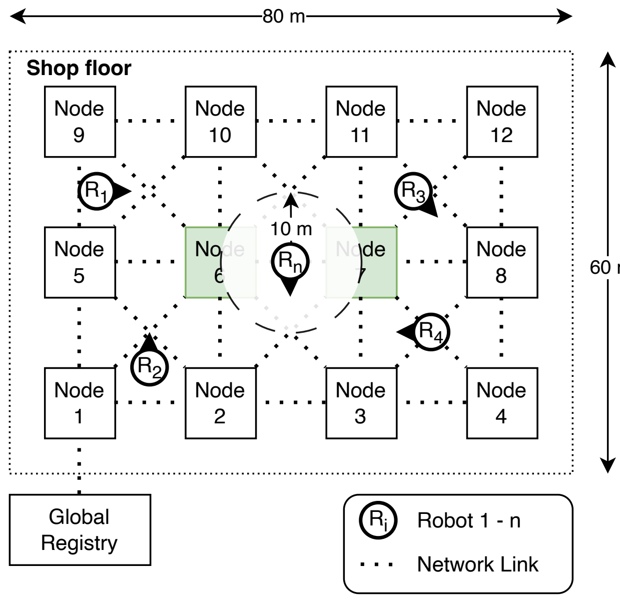

Figure 1. The schematic depicts a factory floor with manufacturing islands that host at least one IoT node (Node i) per cell. Autonomous robots (R_i) can navigate through the formed grid and connect to nodes within a 10 m radius. The *Global Registry* serves as the source of workloads.

1.1 Motivating Example

The railway production setting serves as an illustrative example. The production process is organized around the objective of assembling prepared components. Autonomous robots are constrained in their ability to support this process due to the complex entry and reachability of positions. In contrast, the manufacturing of components can be organized in a cellular manufacturing setting.

The work presented here is limited to an excerpt with stations arranged in a grid pattern to reduce transportation distances and to maintain clear access to protected areas and escape routes. Production islands are equipped with Internet of Things (IoT) nodes for monitoring production metrics in near real-time. Cells may include mechanical assembly, welding, lacquering, or other steps, which opens the potential for robotic collaboration across different production steps and supplementary tasks, such as lifting and positioning. In instances where interaction can benefit from additional process information that is not accessible to the robot due to technical limitations, such as the quality, surface or volume structure, strain, torsion or temperature of the workpiece, or information that is not accessible due to spatial or temporal constraints, such as the aforementioned parameters on a different workpiece's location, the quality of previous production steps or information about subsequent production steps, guidance can be provided by IoT nodes. In order for the IoT device to process the information and generate knowledge that can enrich the robot's interaction, it must be instructed on how to evaluate its sensor data in relation to the robot's current task.

Figure 1 depicts the 80 m by 60 m shop floor area. It is assumed that each manufacturing cell contains at least one IoT node. All nodes are connected to one another using a stable industrial WiFi mesh, forming an interconnected grid. Robots can connect directly to a node when they enter a 10 m perimeter around the node. Workload can be loaded from the registry, which is connected to exactly one node. More details on the setup are given in section 5. Without loss of generality, we refer to our mobile agents as *robots* for use-case and consistency reasons. However, later presented approaches apply to a large number of dynamic scenarios.

1.2 Problem Statement

Although general-purpose robots can access process data using state-of-the-art protocols, the actor logic must be adapted each time, which is a significant drawback. This paper addresses the problem of integrating task offloading in an industrial factory floor environment. We identified the following challenges:

Challenge 1: Heterogeneous & constrained computing platforms. Due to the bespoke relationship between sensor nodes and their processes, hardware platforms are inherently heterogeneous, encompassing a vast array of architectures and hardware capabilities. The price sensitivity of IoT devices often results in the use of microcontrollers with limited processing power and very little memory. Consequently, integrating state-of-the-art offloading and orchestration software is often not possible. ***Our approach:*** We employ the novel bytecode format WebAssembly (WASM), which enables the virtualization of architectures. We modify WebAssembly Micro Runtime (WAMR)[1] for use on tiny devices and present a novel approach to stateful intermittency and subsequent continuation of WASM programs.

Challenge 2: Spatial isolation & untrusted code. When offloading computational tasks to the IoT platforms, it is imperative that the target devices are able to execute their workload without being affected in any way. This necessitates the use of physical isolation, which should be implemented in a manner that minimizes overhead. Furthermore, even in closed environments, code can inadvertently introduce bugs that cause the device to fail. Due to the tight integration with production process control, this can lead to product quality degradation or even product failure, which is unacceptable. ***Our approach:*** We leverage WASM's isolation capabilities in conjunction with a second core for asynchronous execution of workloads, allowing for execution with nearly no impact on the system.

[1] https://github.com/bytecodealliance/wasm-micro-runtime

Challenge 3: Rapid reconfiguration & dynamic loading. From the perspective of the IoT node, work is initiated whenever the robot enters the communication radius. The specific nature of the work is unknown, as it depends on the robot's characteristics, even though it is constrained by the IoT node's capabilities. Nevertheless, distinct workloads must be executable and quickly reconfigurable. ***Our approach:*** Based on the dependency of the node's capability on the workloads and the relation of the manufacturing process to the possible workloads we provide automatically accumulating local subsets of modules that are commonly used in the problem domain.

Challenge 4: Communication overhead & uneven distribution. As the size of transmitted data increases due to the establishment of ML/AI classifiers and the spread of Time Sensitive Networks (TSN) reserving transmission channels, the available wireless communication bandwidth is shrinking. Along with sophisticated communication protocols that reduce communication overhead and improved evaluation algorithms that reduce size of results, 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 again allows for exploiting local dependency, thereby reducing the load on *Global Repository* nodes. Furthermore, using transparent partial updates alleviates the burden on the network.

1.3 Contribution and Outline

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

1. This paper demonstrates how related manufacturing processes can be leveraged to split programs into smaller subroutines, enabling efficient local and nearest-neighbor caching and smoothing unequal load on the central nodes.

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 provide extensive measurements obtained from our real-world test bench and our Nimblenet(-lite) implementation, designed to operate in highly constrained devices with only a few kilobytes of random-access memory (RAM). Furthermore, additional simulation results are presented to supplement these findings.

The remainder is organized as follows: In Section 2, we present potential solutions from the literature and discuss why they are not sufficient. In Section 3, we present our system model and relevant background information. Section 4 outlines 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.

Dynamic updating. To fully leverage the capabilities of IoT nodes, it is necessary to modify the node's program code. This process is analogous to the well-researched and practically relevant field of device updates. However, typical solutions for IoT devices are developed from a top-down approach, assuming that one device actively pushes updates in infrequent cycles. Approaches are tailored to firmware updates, which provide one new, monolithic binary, the typical method of distributing microcontroller software. Consequently, research is focused on authentication, integrity, freshness and validity, which represents a distinct area of interest [7, 9, 20, 26, 27, 39]. Moreover, many of these solutions are vendor-specific, such as Nordic's DFU [3] or Texas Instruments' OAD [4], or tailored to a specific operating system, e.g., Deluge [22] and Sparrow [8]. Package manager solutions, such as YUM [5], APT [1], or TUF [15, 40], utilized by Google Fuchsia, employ techniques, including dependency resolving, which is costly but not necessary for our purposes. Furthermore, this approach necessitates using advanced hardware and often file support or a Linux operating system, which is unavailable on constrained devices.

Workload management across devices. 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 [11] 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 [35], Kreutzer et al. [25], and Nieke et al. [34] have developed frameworks that facilitate strong and weak migration, whereby the application's state is transmitted or left out, respectively. The solutions address additional research questions, such as leveraging service-oriented architectures or improving security. Li et al. [29] suggest that single functions be offloaded to the edge, with static annotations in the source code. Ouacha [36] optimizes the OLSR protocol for the transmission of VMs. Cloud4IoT [37] provides an execution-format independent solution based on Kubernetes agents and OpenStack-based middleware, which is similar to Benomar et al. [14] in the latter. Ada-Things [46] provides insights into load balancing for VM migration, adapting the migration method by analyzing memory page modifications. However, all of these solutions are designed to execute their workload in the cloud, edge cloud, network infrastructure devices, or fog, with the expectation that at least tens of MBs of RAM will be available, and often a Linux operating system. As a result, they are not applicable to IoT nodes due to the constraints imposed by the limited resources. Furthermore, scheduling in a multi-node system is not addressed by any of the works. Yousafzai et al. [48] present a process-based migration that requires a compatible process model, namely a Linux operating system and similar architectures. Wu et al. [47] present an approach where nearby IoT nodes are taken into account as an offloading target. To execute code on heterogeneous architectures, the prototype employs HQEMU [21], a retargetable dynamic binary translation based on QEMU and LLVM. This approach, in conjunction with the costly memory remapping, necessitates the execution on more powerful edge devices. Evaluations conducted on devices with several GBs of RAM substantiate this assertion. Dynamic linking approaches, such as LLL [33], Zephyr's LLEXT [6], Contiki's dynamic loader [2], or custom approaches, are complex, prone to errors, and highly architecture and OS specific.

Distributed data management. In unstructured peer-to-peer overlay networks, data is distributed in a manner that is not dependent on the intrinsic properties of the data itself. This offers insights into potential routing algorithms. Nevertheless, algorithms are inherently incapable of execution and orchestration, including all the challenges those topics introduce. Furthermore, the objective of caching is typically the routing targets rather than the data itself. Still, insights can be provided into this topic from a distribution perspective, as it may be related. The three most-known algorithms are Freenet [16], which leverages a depth-first search combined with a least recently used (LRU) cache; Gnutella [42], which uses a flooding technique with a specified radius; and BitTorrent [23], which depends on a centralized repository for tracker nodes, which are responsible for monitoring the availability of files on individual nodes. The KaZaA [28] protocol employs a hybrid model comprising super-peers, which employ Gnutella requests each. Moreover, delay-tolerant networking protocols [13] are similarly concerned with the efficient delivery of data in intermittent settings. Here, caches are often employed for the store and forward functionality of different routing protocols. However, in our case, this results in a suboptimal cache utilization on single nodes due to tasklets stored for the purpose of store and forward being often irrelevant for the forwarding node itself. Nearest-neighbor caching [32, 45] is a strategy used in distributed systems to improve data access efficiency by caching data close to where it is most frequently accessed. From a purely theoretical standpoint, we are grappling with the same fundamental issue, albeit on a much smaller scale. Consequently, the proposed solutions and heuristics appear to be overly complex and unnecessary.

.

3 System Model and Background

In the following section, the system model and necessary definitions are introduced.

3.1 Hardware Platform

We consider highly constrained devices, particularly those utilized for monitoring tasks, may possess sophisticated external measurement hardware yet remain constrained in their available computing (i.e., few cores) and memory capacities. This is primarily attributed to cost considerations, although some devices may also be designed to operate within a limited energy budget due to their battery-powered nature.

This paper examines the general applicability of our approach to said constrained devices located on the extreme edge, where software orchestration and execution can be conducted in concurrently. Nodes are interconnected in a mesh topology, with some nodes directly connected to repository nodes, as illustrated in Figures 2 and 1.

3.2 Software Framework

Without loss of generality, we mainly base our work on WebAssembly (WASM), a new bytecode format initially introduced to the browser environment but has been adapted to IoT. WASM provides a low bytecode size combined with near native-speed execution. Hardware details are abstracted, making WASM executable on various devices. As the language is constructed as a compile target and utilizes well-established compiler frameworks, i.e., LLVM, it can be compiled from various source languages and with enhanced optimizations. Further, WASM's bytecode assures memory isolation by combining a restrictive typing system with runtime checks. Interaction with the host system is possible via WASM's *import* feature, registering native function callbacks as executable. [18, 44]

We further assume that, as is typical in the industrial domain, applications are composed of many smaller functional units and are only loosely coupled in the sense of cause-effect chains [24]. Well-known examples are the AUTOSAR [12] application components, comprising tasks and runnables, and the implicit register communication [17] used there. Other examples are dividing tasks into subtasks, the actor model [10], i-lets [43], or servlets.

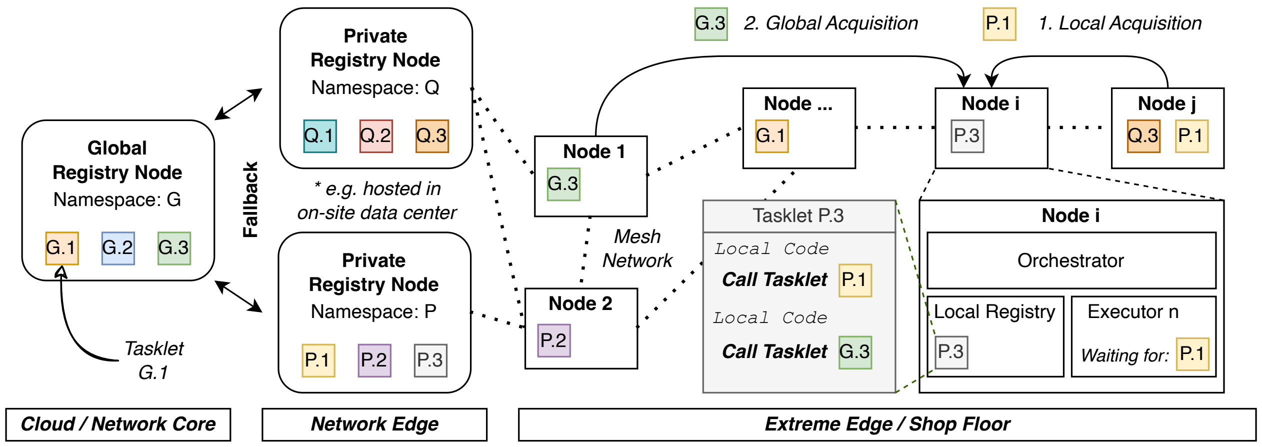

Figure 2. Schematic description of NimbleNet's environment and core architecture: One or more *Global registry nodes* are responsible for providing *Tasklet binaries*, which are grouped by namespaces. Each tasklet may contain calls to other tasklets or local code. *Nodes* are connected in a mesh layout and cache tasklets in their *Local registry*, depending on the tasklet size and their available memory. Additionally, each node contains an *Orchestrator* handling transmission and scheduling and an *Executor* to execute the tasklet.

3.3 Tasklets

For our work, we assume that said functional units are small, platform-independent code snippets that are practically self-contained and exhibit an easy-to-estimate execution time. We, therefore, adopted the notion of tasklets introduced by Schäfer et al. [41] in a slightly modified form. In particular, our underlying tasklet executor supports cooperative scheduling, which allows tasklets to call other tasklets, which we will refer to as sub-tasklets, during their execution and relinquish control flow to the scheduler. Note that the analysis and the identification of optimal scheduling strategies for systems is beyond the scope of this work.

3.4 Taskset

In this paper, we refer to sets of tasklets, or tasksets, as the units of analysis in this work. In particular, one set, T_m, serves as a baseline, while the other, T_n, is employed to illustrate the system's advantages. Both sets of tasklets execute the same compute load and are fundamentally identical regarding execution time and binary size of their tasklets. In contrast to T_m, which consists of monolithic tasklets containing all compute load, T_n comprises those same tasklets split into top-level tasklets, or chains, t_f, and sub-tasklets, or algorithms, t_a. Each repeating algorithm t_a is separated into its own tasklet, which can be reused between top-level tasklets t_c. The execution nature and structure of a tasklet can be observed in Figure 3. Details about runtime, dependency, and call behavior are given in section 6.

4 Approach

As previously stated in Section 1, our approach is designed for industrial applications where it is assumed that the workload exhibits a spatial dependency. Subsequent production steps or actions related to permanently installed bodies are logically related and, therefore, mostly in the same problem domain. This suggests that underlying concepts are likely to be shared, and that the same libraries can be used. This motivates the selection of tasklets, as introduced in Section 3. Consequently, NimbleNet is constructed based on tasklets, which allows the combination of shared libraries into sub-tasklets, shared among tasklets. This greatly increases the shared code base, thereby enabling the formation of fine-grained caches per node and the formation of local subsets of frequently used algorithms that can be reused between neighboring nodes.

Figure 2 includes a schematic tasklet, which is a unit of executable code that can contain any number of *local code* sections, interspersed with calls to other sub-tasklets. These sections can be of any executable code, including native instructions, as long as the corresponding *executor* is present on the node. The executor either interprets the local code instructions or, when encountering a tasklet call, requests the sub-tasklet and executes it before continuing with the original one. In theory, any depth of tasklets is possible, although in practice, this is limited by the available memory.

The following paragraphs present a detailed account of the fundamental inner works of NimbleNet. To facilitate comprehension, the subsequent section is organized in accordance with the typical lifecycle of a tasklet, encompassing the stages of acquisition, execution, and offloading/eviction. Figure 2 provides a conceptual overview of NimbleNet.

4.1 Tasklet Acquisition

Before tasklets can be executed, their binary and metadata must be acquired. Initially, tasklets are stored in global and private registries. Those serve as tasklet archives and can introduce new tasklets into the system. Further, this system of independent registries natively allows for hierarchical tasklet organization. This is an advantage, e.g. when, on the one hand, having a globally hosted registry with standard tasklets (e.g., drivers, hardware-specific tasklet) and, on the other hand, having a local registry with confidential or process-tailored tasklets (e.g., local classifiers in a factory environment). To improve load balancing, response time, and overall network load, tasklets can also be loaded directly from all participating nodes that currently hold the respective tasklet.

Preloading. Additionally, it is possible to load all required sub-tasklets of a given tasklet together with its initial acquisition. This increases the initial acquisition time and overall memory footprint; however, the tasklet can be executed subsequently with almost no overhead.

Acquisition algorithm. The acquisition of a tasklet is performed in accordance with Algorithm 1. Initially, it is verified whether the tasklet is already present in the local cache. If this is the case, it is marked as *reserved*, signifying that the tasklet is required during subsequent execution and must not be offloaded. If the tasklet cannot be located in the local cache, it must be acquired from global/private registries or other nodes.

As previously stated, for load balancing and response time purposes and to decrease overall network load, it is desirable to acquire the tasklet from a node as close as possible to the requesting node. Therefore, the acquisition process is divided into three steps. Initially, a tasklet request is broadcasted to the direct neighbors of the respective node (one hop away, TTL = 1). Only responses indicating the availability of a tasklet and timeouts are considered, as it is irrelevant whether a node lacks the tasklet or is not present in dynamic environments.

In the event that no direct neighbor has the requested tasklet in its local cache, a second global broadcast is initiated following the expiration of the local timeout. The node that responds with the fastest latency is selected as the *winner*. This approach has several advantages. Firstly, it reduces overall response time by eliminating the need to wait for a specific timeout. Secondly, it preserves locality in mesh networks by ensuring that responses from closer nodes with similar hardware reach us more quickly than those from more distant nodes.

Should both acquisition attempts fail, as a fallback, the corresponding global/private registry is via a direct call.

Upon identifying a node with the given tasklet, the exchange of tasklet metadata ensues. This metadata encompasses, among other elements, the tasklet size and hash. The local cache is then evaluated to ascertain its capacity to accommodate the tasklet's binary. In the event that this capacity is insufficient, other tasklets that are not required are evicted. Subsequent to this evaluation, the tasklet binary is exchanged, stored in the local cache, and marked as *reserved*.

Algorithm 1 Tasklet acquisition

```
Input: $id$                                              ▷ Tasklet identifier
Output: None | tasklet                                   ▷ The requested tasklet
 1: $attempts \leftarrow 5$
 2: $hops_n \leftarrow 1$
 3: $tout_n \leftarrow 0.1$
 4: $hops_a \leftarrow 255$
 5: $tout_a \leftarrow 0.5$
 6: while $attempts \neq 0$ do
 7:     $attempts = attempts - 1$
 8:     if $t\_id \in cache$ then                        ▷ Check cache
 9:         cache.reserve(id)                            ▷ Reserve tasklet
10:         return $cache[id]$
11:     end if
12:     src = gather($id$,$hops_n$, $tout_n$)            ▷ Local search
13:     if src == None then
14:         src = gather(id,$hops_a$, $tout_a$)          ▷ Global search
15:     end if
16:     if src == None then
17:         src = gather_fallback(t_id)                  ▷ Fallback
18:     end if
19:     if src == None then
20:         continue
21:     end if
22:     tasklet_meta = get_metadata(src)                 ▷ Get metadata
23:     if tasklet_meta.size > cache.free_size then
24:         success = cache.free(tasklet_meta.size)      ▷ Evict
25:         if not sucess then
26:             continue
27:         end if
28:     end if
29:     tasklet = get_tasklet(src)                       ▷ Fetch tasklet
30:     cache.add($id$, tasklet_meta, tasklet)
31:     cache.reserve($id$)                              ▷ Reserve tasklet
32:     return tasklet
33: end while
34: return None                                          ▷ Tasklet could not be acquired
```

4.2 Tasklet Execution

Once a tasklet has been acquired, it can be executed by a local executor instance. Different executors support different

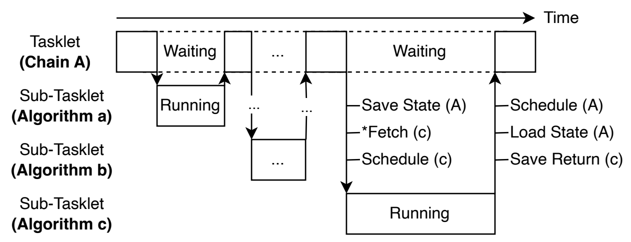

Figure 3. Exemplary tasklet chain execution on a single core: The tasklet (Chain A) is comprised of three sub-tasklets (algorithms, a, b, and c). The execution of the tasklet continues until an algorithm is reached, at which point execution is transferred to it. The state of A is saved, the algorithm is fetched and executed, and then execution is returned to Chain A.

programming languages and intermediate representations, allowing for the transparent mixing of programming languages between a tasklet and its sub-tasklets. This provides the option to incorporate device-specific, hardware-tailored sub-tasklets, if needed. Furthermore, tasklets support cooperative scheduling. In a simple case, as illustrated in Figure 3, the top-level tasklet serves as a *chain*, combining multiple sub-tasklets, e.g., signal processing algorithms, into a single execution unit. In this case, after *Chain A* is initiated, it transfers control flow to other sub-tasklets (*Algorithm a*, *b* and *c*) and awaits their results. In particular, when only a single executor is available, context switching must occur between tasklets. The typical procedure commences with the local saving of the state of the currently running tasklets. In the event that this has not already been done, the new tasklet must then be acquired from another node. Subsequent to this, the binary of the called tasklet is loaded and the corresponding executor is initialized with the arguments of the calling tasklet and the binary of the new sub-tasklet.

In our implementation, where tasklets are expressed in WASM binaries, the actual execution is carried out by the WAMR. Sub-tasklet execution leverages WASM's import feature to transfer execution to the host system. We employ a wrapper function to each tasklet call, i.e., native call. The native handler will store the parameters it is called with and the sub-tasklet id to call together with the current state of the execution.

The execution state is completely represented by the WASM call and operand stacks containing all local variables, the WASM heap, and the runtime's state after initialization. This state is saved in order to improve recovery speed, as it avoids the need to re-parse the binary and create instances of static objects, such as populating function reference tables, loading initial values of globals, and binding native addresses to WASM functions. Subsequently, the native handler transitions to an error state, immediately terminating the WASM execution.

Upon the initiation of the sub-tasklet execution, the WAMR executor utilizes the previously allocated memory area to prevent heap fragmentation, a crucial consideration on constrained devices. The stored sub-tasklet arguments are loaded and execution is initiated.

To restore the top-level tasklet, the saved WASM state is loaded. The runtime's configuration is applied, and the operand stack, the heap, and global values are replaced with those from the initial state. The call stack is modified by removing the last stack frame and decrementing the instruction pointer to restart the sub-tasklet call, i.e., the native call instruction. By providing the results from the preceding sub-tasklet execution on invocation, the native handler will promptly return those values in lieu of entering an error state.

4.3 Tasklet Eviction

Once a tasklet has completed its execution, its *reservation flag* is revoked. It is then up to the eviction strategy to determine whether the tasklet should remain in the local cache or be evicted. However, this eviction strategy is highly dependent on the system's environment.

In many cases, straightforward eviction strategies, such as least recently used (LRU), least frequently used (LFU), or hybrid variants like adaptive replacement cache (ARC)[31], are sufficient. However, the optimal eviction strategy is highly dependent on additional features, including tasklet size, last acquisition time, and the number of neighboring nodes. Therefore, it is essential for each node to learn optimal eviction patterns to incorporate these features and more complex environment-dependent access patterns efficiently. Nevertheless, for the purposes of this paper, we demonstrate the applicability of the overall approach, even when utilizing simple eviction strategies like, in our case, simple LFU caches. Consequently, the subject of sophisticated eviction strategy optimization is not addressed in this work and may be the subject of future research.

4.4 Additional Advantages

Even if not further investigated in this paper, the presented approach has many more application areas, especially in the extreme edge.

Serverless computing. Due to NimbleNet's highly sandboxed design, tasklets can be transparently executed on other nodes with the current approach. This opens up the possibility of efficient task offloading and overall serverless computing in the extreme edge.

In-network computing. Especially In-network computing, where data is (pre)processed on the way to its destination,

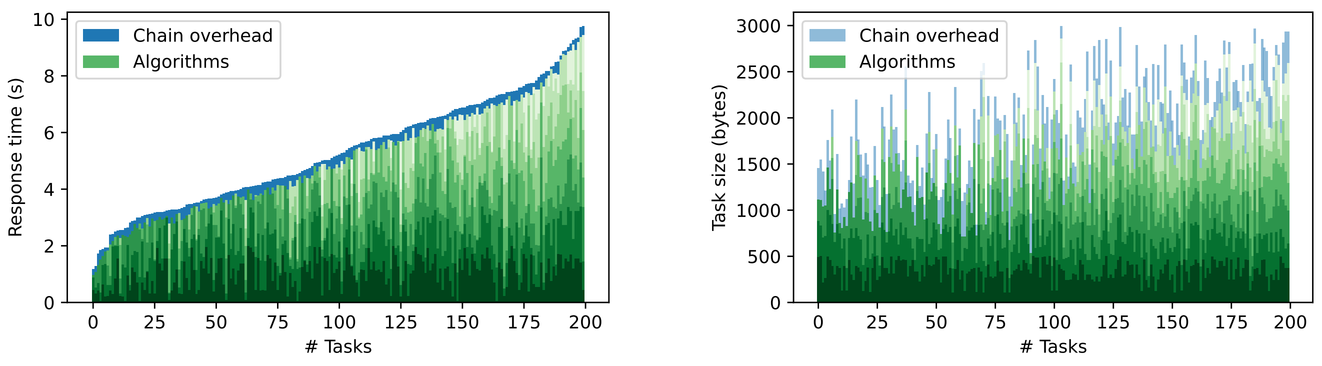

Figure 4. Evaluation taskset, sorted by response time. The left graph shows the response time, the right graph shows the tasklet's size

routing nodes can highly profit from fast reconfiguration times and small network overhead. Especially in this scenario, it is highly probable, that *adjacent* nodes share the same main workload due to load balancing and overprovisioning. Further, when opening up in-network computing to the public, code has to be delivered from multiple global code repositories and has to be computed in a sandboxed way.

On-demand loading. Due to NimbleNet's design, especially for the proposed data processing chains, it is possible due to dynamic loading/unloading of sub-tasklets to execute chains with an overall larger binary size than the system's internal cache. However, the system's response time increases due to tasklets not being preloaded in this case.

5 Experimental Setup

In this section, we outline the overall experimental setup.

5.1 Scenario

Our experimental setup follows our use case, which we already presented in Section 1.1. In our scenario, multiple robots move pseudorandomly through a factory floor, triggering different sensing tasks on nearby constrained devices in order to enhance their perception of the environment.

During the simulation, the movement patterns and the thereof resulting list of called tasklets are generated for deterministic reasons. During evaluation on the testbed, these call patterns are then played back on the real testbed in order to simulate the tasklet calls from the mobile agents on the factory floor. This ensures that the same tasklet calls will be observed in simulations and on the testbed.

5.2 Simulation

Prior to execution on our testbed, a scenario is first simulated on our discrete-event (network) simulator. The simulator is based on the NetworkX [19] and SimPy [30] libraries.

The simulator represents the number and types of tasklet executors per node, as well as memory and CPU constraints. However, it is assumed that networking operations (receiving, sending, and forwarding packages) can occur in parallel with the actual orchestration and execution of tasklets. Additionally, the raw response time of tasklets in the simulation is adjusted to reflect the computational capabilities of the target platform. Consequently, the actual response time for given tasklets is initially measured on the target platform prior to the system being simulated. The simulation is intended to provide an initial estimation of the performance of the system and to test and compare different protocols. Currently, complex interactions and side effects, especially for highly constrained systems that only utilize one core for network communication and orchestration, are not simulated due to their highly dependent nature on the system's implementation details.

The simulation setup is oriented towards the motivating example presented in Section 1.1. The 80 m by 60 m area is simulated with an interconnected node grid, hosting sensor nodes spaced 20 m apart as depicted in Figure 1. The *Global Registry* is connected to *Node 1*. Continuous pseudo-random movement patterns are generated for the autonomous robots using continuous noise functions. An illustrative example of a robot trajectory is presented in Figure 5. Upon the robot's entry into the node's 10 m radius, a connection is establishable, and the robot initiates the execution of its task chains on that node at regular, uniformly distributed intervals. The network traffic and the execution on the node are then simulated. Moreover, the initiated tasks are documented during

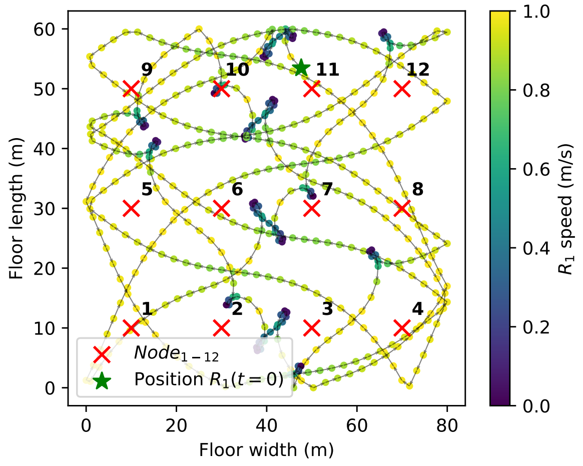

Figure 5. Exemplary robot 1 trace over 1800 s on our 80 m × 60 m shop foor area together with IoT node positions

the simulation and subsequently reproduced during the evaluation on real hardware testbed.

5.3 Taskset Generation

Without loss of generality, the response time of tasklets in our use case is mainly CPU-bound and network-bound. For later evaluation, two synthetic tasklet sets are generated.

Nimble tasklet set. The initial *nimble* taskset T_n comprises task chains in the form of top-level tasklets combined with sub-tasklets. These showcase individual but shareable algorithms. During the generation process, all tasklet sizes and response times are drawn independently and uniformly. Subsequently, the algorithms representing the sub-tasklets are randomly and uniformly distributed to the top-level tasklets. Furthermore, additional *response time* is distributed uniformly between tasklet calls to reach the specified total response time, simulating additional data pre- and post-processing performed by the top-level tasklet before and after calling further algorithm sub-tasklets.

Monolithic tasklet set. Secondly, all top-level tasklets from the *nimble* tasklet set are converted into *monolithic* tasksets T_m by inlining the algorithm code taken recursively from the respective sub-tasklets. Consequently, they form multiple large top-level only tasklets while maintaining the same characteristics in terms of runtime and total binary size.

An exemplary overview of the tasklet-chain composition, its response time, and tasklet size distribution can be observed in figure 4.

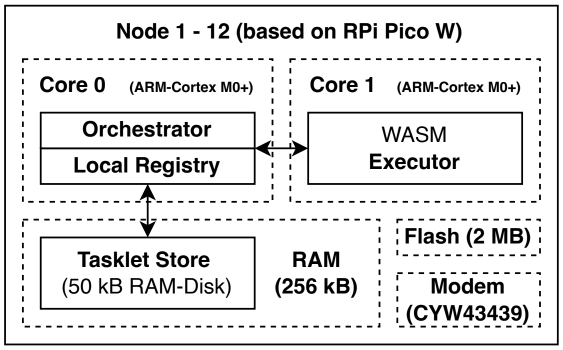

Figure 6. A node's system implementation overview: Core 0 hosts the orchestrator and the registry, while Core 1 is exclusively dedicated to the WASM Executor. Tasklets are stored in RAM, while the flash contains static code.

5.4 Testbed

As illustrated in figure 1, our testbed comprises of twelve constrained nodes and a thirteenth registry node, which is initially utilized for tasklet storage.

Node hardware. The Raspberry Pi Pico W[38] was selected as the baseboard for the twelve highly constrained nodes to align with the hardware platform requirements outlined in section 3. This device was chosen due to its constrained CPU (ARM Cortex M0+ operating at 133 MHz) and limited memory (256 kB RAM). Additionally, its two CPU cores facilitate the execution of tasks in parallel.

The tasklet binaries are stored in a 50 kB RAM disk as depicted in figure 6. All additional tasklet meta-information is stored and managed outside the RAM disk. All software components required for orchestration and tasklet acquisition run on Core 0, enabling exclusive tasklet execution on Core 1.

Network. A semi-virtual software-based networking approach has been selected for the testbed, offering rapid reconfigurability and reproducibility. All nodes are connected to a central access point, which serves as the gateway for all traffic between nodes. Sophisticated routing rules are employed on the access point to ensure that each node can only reach the neighbors it is connected to in our network structure, thus emulating the desired topology. The access point facilitates the transparent forwarding of packets to the subsequent node on a designated route within the emulated topology.

In the emulated network, nodes perform the same forwarding operations for unicast and broadcast operations as they would in the real network. Figure 7 depicts an exemplary message forward pass. As the objective is to compare T_n and T_m on their relative acquisition time and memory overhead only, this represents an ideal compromise between a real-world hardware setup and fast reconfigurability.

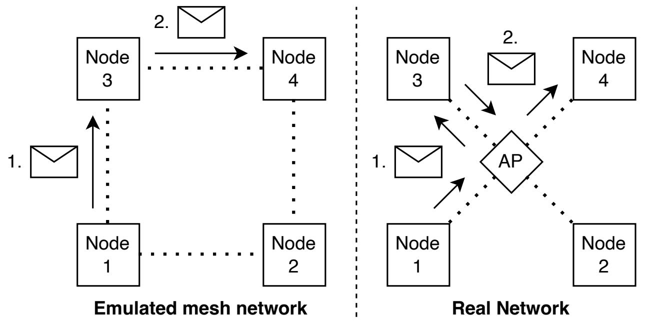

Figure 7. Message forwarding in the real vs. the emulated network utilizing a central access point

Table 1. Setup parameters

Parameter	Min	Max
Duration (s)	3600	
# Chains	200	
Chain RT (s)	0.1	0.5
Chain Mem (bytes)	100	500
# Algorithms	50	
Algorithm RT (s)	0.01	2
Algorithm Mem (bytes)	100	500
Algorithms per Flow	3	7
# Robots	20	
# Chains per Robot	20	
Trigger Interval (s)	9	11
Robot speed (m/s)	0	1
Cache Size (bytes)	32768	

6 Evaluation and Results

In this section, NimbleNet is evaluated initially through simulation and subsequently on the testbed. First, six simulations with parameters as outlined in Table 1 are executed. Thereafter, the same generated taskset is executed on the real-world testbed. For the purposes of this analysis, we have chosen to compare the different response times, network load, and network load distribution in order to gain an initial impression of the advantages and limitations of the system on the chosen testbed. Furthermore, the discrepancies between the simulation and testbed results will be discussed.

6.1 Evaluation Parameters

As illustrated in Table 1, we adhere to the original configuration of 20 robots and 12 nodes, with one registry connected directly to node 1. Without loss of generality, we further assume that each agent attempts to initiate a task chain, i.e., a top-level tasklet, every 9 - 11 s on an arbitrary node in its immediate vicinity (10 m). Each tasklet chain combines three to seven algorithm sub-tasklets. For algorithm sub-tasklets, we assume a response time between 0.01 and 2 s to accommodate both short-running tasklets and relatively long-running tasklets, including complex classifiers. Similarly, the algorithm size lies within our case between 100 and 500 bytes. This encompasses algorithms of varying complexity, from simple sorting and signal processing to more memory-intensive algorithms such as small deep neural network (DNN) classifiers. Overall, we observe an expected average algorithm size and response time of 300 bytes and $\approx$ 1 s, respectively. For the complete tasklet chain, we obtain an average expected size and response time of 1800 bytes and $\approx$ 5.3 s, respectively, not including overheads due to tasklet acquisition and calls. Consequently, the same response time and size estimates are obtained when converting the tasklet chains into monolithic tasklets. In particular, the tasklet sizes were chosen to be relatively small in order to demonstrate that even with small algorithms, NimbleNet still outperforms the monolithic approach.

Tasklet share. As anticipated, when tasklets are not reused between chains, NimbleNet performs slightly worse than the monolithic approach due to additional communication overhead. However, the load-balancing advantages remain. Conversely, reusing almost all tasklets between all chains leads to NimbleNet vastly outperforming the monolithic approach. This, however, is not a realistic scenario. Therefore, we proceeded with a more realistic scenario in which 200 distinct chains were established, each sharing 50 distinct algorithms. Each robot was assigned 20 random chains, resulting in a balanced distribution of shared and unique flows between robots.

Total tasklet calls. As previously stated in this evaluation scenario, agents move at speeds between 0 and 1 m/s due to the potential for carrying heavy loads and due to safety restrictions imposed by the environment. Given the nature of this scenario, there exist instances where agents have no direct contact with any node. This phenomenon occurs in $\approx$23,% of all tasklet calls, resulting in a reduction in the total number of executed chains/monolithic tasklets from $\approx$ 7200 down to $\approx$ 5470.

6.2 Network Traffic Distribution

Figure 8 depicts the total network traffic, the sum of sent (TX) and received (RX) bytes, of each individual node running both NimbleNet and monolithic programs for one hour. The left graph presents the simulation results, while the right graph displays the results from our real-world testbed. The measurements demonstrate that the sent bytes for node 1 are reduced by a factor of four and five for the simulation and test-bed measurements, respectively, when comparing the nimble and monolithic approaches. Overall, as can be seen, the total network traffic is more evenly distributed between all nodes. This illustrates the dispersion of the source for programs when queried. The data received in both cases is

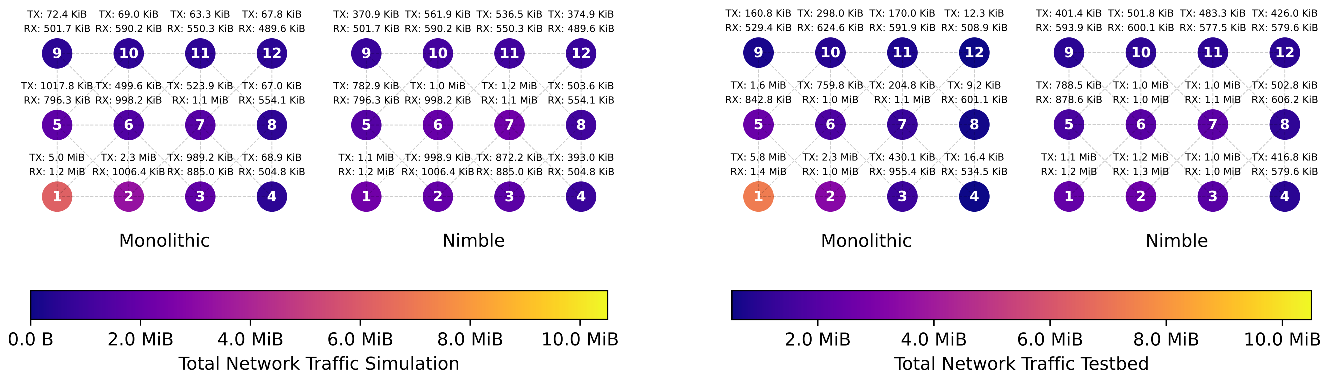
TX: 72.4 KiB RX: 501.7 KiB
TX: 69.0 KiB RX: 590.2 KiB
TX: 63.3 KiB RX: 550.3 KiB
TX: 67.8 KiB RX: 489.6 KiB
9
10
11
12
TX: 1017.8 KiB RX: 796.3 KiB
TX: 499.6 KiB RX: 998.2 KiB
TX: 523.9 KiB RX: 1.1 MiB
TX: 67.0 KiB RX: 554.1 KiB
5
6
7
8
TX: 5.0 MiB RX: 1.2 MiB
TX: 2.3 MiB RX: 1006.4 KiB
TX: 989.2 KiB RX: 885.0 KiB
TX: 68.9 KiB RX: 504.8 KiB
1
2
3
4
Monolithic
TX: 370.9 KiB RX: 501.7 KiB
TX: 561.9 KiB RX: 590.2 KiB
TX: 536.5 KiB RX: 550.3 KiB
TX: 374.9 KiB RX: 489.6 KiB
TX: 782.9 KiB RX: 796.3 KiB
TX: 1.0 MiB RX: 998.2 KiB
TX: 1.2 MiB RX: 1.1 MiB
TX: 503.6 KiB RX: 554.1 KiB
TX: 1.1 MiB RX: 1.2 MiB
TX: 998.9 KiB RX: 1006.4 KiB
TX: 872.2 KiB RX: 885.0 KiB
TX: 393.0 KiB RX: 504.8 KiB
Nimble
0.0 B
2.0 MiB
4.0 MiB
6.0 MiB
8.0 MiB
10.0 MiB
Total Network Traffic Simulation
TX: 160.8 KiB RX: 529.4 KiB
TX: 298.0 KiB RX: 624.6 KiB
TX: 170.0 KiB RX: 591.9 KiB
TX: 12.3 KiB RX: 508.9 KiB
TX: 1.6 MiB RX: 842.8 KiB
TX: 759.8 KiB RX: 1.0 MiB
TX: 204.8 KiB RX: 1.1 MiB
TX: 9.2 KiB RX: 601.1 KiB
TX: 5.8 MiB RX: 1.4 MiB
TX: 2.3 MiB RX: 1.0 MiB
TX: 430.1 KiB RX: 955.4 KiB
TX: 16.4 KiB RX: 534.5 KiB
Monolithic
TX: 401.4 KiB RX: 593.9 KiB
TX: 501.8 KiB RX: 600.1 KiB
TX: 483.3 KiB RX: 577.5 KiB
TX: 426.0 KiB RX: 579.6 KiB
TX: 788.5 KiB RX: 878.6 KiB
TX: 1.0 MiB RX: 1.0 MiB
TX: 1.0 MiB RX: 1.1 MiB
TX: 502.8 KiB RX: 606.2 KiB
TX: 1.1 MiB RX: 1.2 MiB
TX: 1.2 MiB RX: 1.3 MiB
TX: 1.0 MiB RX: 1.0 MiB
TX: 416.8 KiB RX: 579.6 KiB
Nimble
2.0 MiB
4.0 MiB
6.0 MiB
8.0 MiB
10.0 MiB
Total Network Traffic Testbed

Figure 8. Load distribution simulations monolithic and nimble approach with the simulation results depicted on the left and the tesbed results shown on the right.

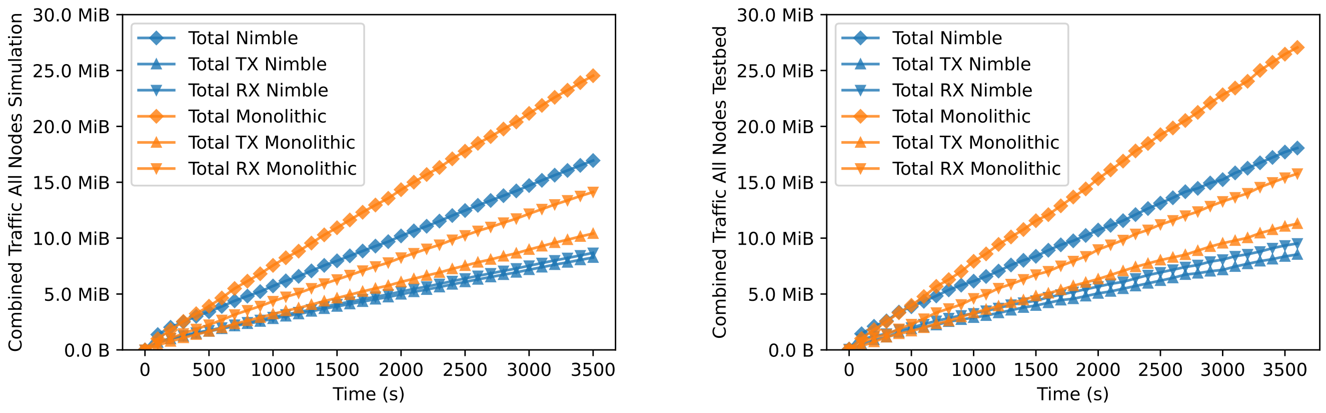

Figure 9. Total network load simulations monolithic and nimble approach six simulations

equivalent due to the identical programs being utilized in both instances. The difference between the simulation and the testbed results can be attributed to two major causes.

Housekeeping packages. In the simulation, the network traffic is exclusively related to the NimbleNet. In contrast, in the real testbed, additional messages are distributed within the network for housekeeping purposes. These include simple ARP, DHCP, and NTP messages, as well as more complex routing protocol messages.

Non determinism. Nevertheless, these housekeeping messages represent a relatively minor component of the overall additional network traffic. In the simulation, connections are always assumed to have no package drop. However, this is not the case in the testbed, where, during package bursts, packages can be dropped due to the limited forwarding capacity of highly constrained nodes. These packages must then be retransmitted, which increases the overall network traffic slightly. The most significant impact on the overall network traffic is the result of non-determinism in the tasklet acquisition process. As previously outlined in section 4, for our greedy tasklet acquisition approach, we assume that the response time for a tasklet acquisition request is highly correlated with the hop distance to the respective node in relation to the requesting node. However, due to the constrained nature of nodes, there are instances where request messages are dropped or not answered in time. Consequently, in the real testbed, the response from nodes situated at a greater distance can reach the requesting node in advance of the response from local nodes, resulting in an increase in overall network traffic. As these effects are observed in both NimbleNet and the monolithic approach, this does not affect the conclusions drawn from the findings.

6.3 Total Network Toad

Figure 9 depicts the total network load over time, with the left graph representing the simulation and the right graph representing the testbed. It can be observed that the cumulative load of the NimbleNet approach is slightly higher in the initial phase. This is due to the additional routing and querying overhead that has been introduced. The break-even point is reached at $t = 300$ s and $t = 400$ s, respectively, for the simulation and the testbed. At this point, the smaller caching units and the nearest neighbor acquisition begin to demonstrate their advantage. Moreover, the observed trend in the simulation can be found in the real-world implementation. As previously stated in section 6.2, the real-world testbed exhibits a slight increase in network traffic compared to the simulation.

6.4 Response Times

NimbleNet's enhanced caching functionality and nearest neighbor acquisition enable faster tasklet acquisition due to successful requests on neighboring nodes. Consequently, the overall response time is reduced. Figure 10 illustrates this phenomenon by depicting the mean response time of all completed tasklets in a sliding window of 100 s. It can be observed that while the raw tasklet execution times are nearly equivalent, the total time for acquisition is faster for the NimbleNet approach compared to the monolithic approach. This phenomenon is primarily caused by the long hop delay of 100 ms per hop in this mesh scenario. However, when limited by link speed (e.g., when using larger tasklets), the same effect occurs. Further, the marginal discrepancy in execution time for NimbleNet of ≈30 ms on average is attributable to the necessity of executing a greater number of code loadings and setups. Furthermore, the advantage of caching is corroborated by the previous measurement, which shows the same break-even point after 300 and 400 s.

Figure 10 illustrates that both the simulation and the actual execution on the testbed demonstrate a similar initial pattern, where the overall response times significantly increase until the majority of the tasks are propagated into the system itself. After approximately 400 s for the testbed, the response time improvements resulting from the nearest neighbor acquisition and local caching become evident, and NimbleNet outperforms the monolithic approach most of the time.

7 Conclusion

This paper presents NimbleNet, a lightweight and platform-independent distribution and orchestration approach targeting highly constrained devices attempting to reuse existing, process-tailored IoT devices to assist general-purpose robots in cellular manufacturing processes. The paper demonstrates how to address heterogeneous device architectures and spatial isolation. It also utilizes the process dependency to employ efficient nearest-neighbor caching for program acquisition. We present a novel approach to intermittent execution of WASM programs and to save and restore program state. Extensive evaluation results were obtained from a network simulation and confirmed by a real-world testbed implemented on Rasbperry Pi Pico W microcontrollers with only a few hundred kilobytes of RAM.

The results of the measurements demonstrated that the central node experienced a reduction in communication load and that communication among the nodes became more equal. Moreover, the network usage was generally lower and the response times were faster than those observed in the monolithic approach. These findings indicate that NimbleNet is well-suited to address the challenges presented and to fulfill the requirements of the use case.

References

[1] Apt: Advanced Package Tool. https://wiki.debian.org/PackageManagement.

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

[3] Nordic Semiconductor Device Firmware Update. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Flib_bootloader_dfu_process.html.

[4] Texas Instruments Over the Air Download (OAD). https://software-dl.ti.com/lprf/sdg-latest/html/oad-ble-stack-3.x/oad.html.

[5] YUM: Yellow Dog Updater. http://yum.baseurl.org/.

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

[7] 2012. LwM2M. http://openmobilealliance.org/release/LightweightM2M/V1_0-20121127-C/OMA-AD-LightweightM2M-V1_0-20121127-C.pdf.

[8] 2022. Sparrow. https://github.com/sics-iot/sparrow.

[9] 2024. mcumgr. The Apache Software Foundation.

[10] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT press.

[11] 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

[12] AUTOSAR. 2013. Specification of Operating System (Version 5.1.0).

[13] Fatima Zohra Benhamida, Abdelmadjid Bouabdellah, and Yacine Challal. 2017. Using delay tolerant network for the Internet of Things: Opportunities and challenges. In *2017 8th International Conference on Information and Communication Systems (ICICS)*. 252–257. https://doi.org/10.1109/IACS.2017.7921980

[14] 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/

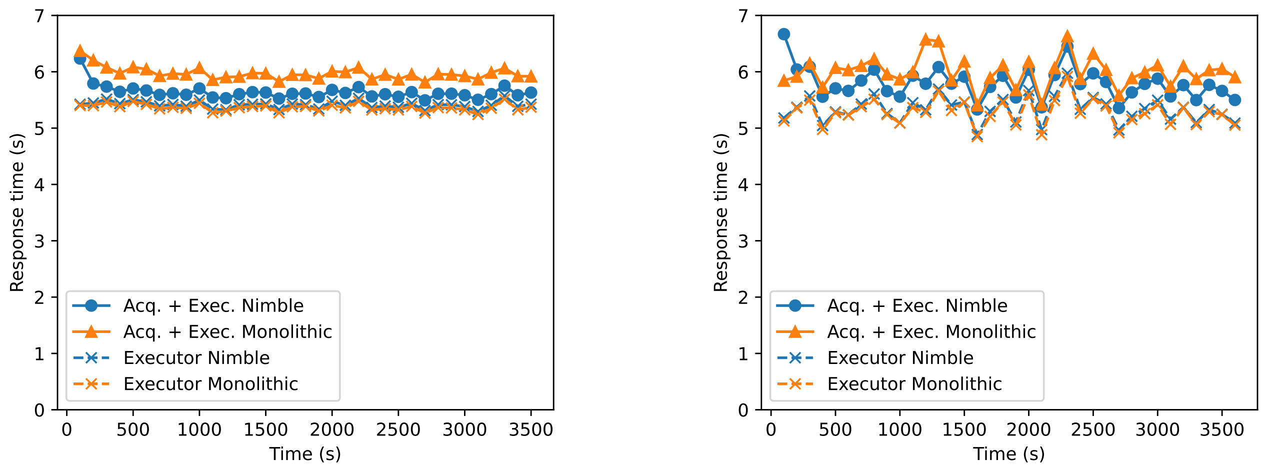

Figure 10. Response time testbed monolithic and nimble approach (sliding window 100 s)

10.1145/3380955

[15] Justin Cappos, Trishank Karthik Kuppusamy, shua Lock, ina Moore, and Lukas Pühringer. 2023. The Update Framework Specification. https://theupdateframework.github.io/specification/v1.0.33/.

[16] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. 2001. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, Hannes Federrath (Ed.). Springer, Berlin, Heidelberg, 46–66. https://doi.org/10.1007/3-540-44702-4_4

[17] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. 2010. Scheduling Dependent Periodic Tasks without Synchronization Mechanisms. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. https://doi.org/10.1109/RTAS.2010.26

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[19] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[20] José L. Hernández-Ramos, Gianmarco Baldini, Sara N. Matheu, and Antonio Skarmeta. 2020. Updating IoT Devices: Challenges and Potential Approaches. In *2020 Global Internet of Things Summit (GIoTS)*. 1–5. https://doi.org/10.1109/GIOTS49054.2020.9119514

[21] 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

[22] Jonathan W. Hui and David Culler. 2004. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*. Association for Computing Machinery, New York, NY, USA, 81–94. https://doi.org/10.1145/1031495.1031506

[23] IAB and Gonzalo Camarillo. 2009. *Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability*. Request for Comments RFC 5694. Internet Engineering Task Force. https://doi.org/10.17487/RFC5694

[24] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2015. Real World Automotive Benchmarks for Free. In *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[25] 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.

[26] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. 2019. UpKit: An Open-Source, Portable, and Lightweight Update Framework for Constrained IoT Devices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2101–2112. https://doi.org/10.1109/ICDCS.2019.00207

[27] Teemu Laukkarinen, Lasse Määttä, Jukka Suhonen, Timo D. Hämäläinen, and Marko Hännikäinen. 2011. Design and Implementation of a Firmware Update Protocol for Resource Constrained Wireless Sensor Networks. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)* 2, 3 (2011), 50–68. https://doi.org/10.4018/jertcs.2011070103

[28] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. 2003. Deconstructing the Kazaa Network. In *Proceedings the Third IEEE Workshop on Internet Applications. WIAPP 2003*. 112–120. https://doi.org/10.1109/WIAPP.2003.1210295

[29] 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

[30] Norm Matloff. 2008. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2, 2009 (2008), 1–33.

[31] Nimrod Megiddo and Dharmendra S Modha. 2003. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference*

on File and Storage Technologies (FAST 03).

[32] Hassan Naderi Mohammad Reza Abbasifard, Bijan Ghahremani. 2014. A Survey on Nearest Neighbor Search Methods. *International Journal of Computer Applications* 95, 25 (June 2014), 39–52. https://doi.org/10.5120/16754-7073

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] Raspberry Pi Ltd. Raspberry Pi Pico and Pico W. https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

[39] Kristina Sahlmann, Vera Clemens, Michael Nowak, and Bettina Schnor. 2021. MUP: Simplifying Secure Over-The-Air Update with MQTT for Constrained IoT Devices. *Sensors* 21, 1 (Jan. 2021), 10. https://doi.org/10.3390/s21010010

[40] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable Key Compromise in Software Update Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/1866307.1866315

[41] Dominik Schafer, Janick Edinger, Justin Mazzola Paluska, Sebastian VanSyckel, and Christian Becker. 2016. Tasklets: "Better than Best-Effort" Computing. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. 1–11. https://doi.org/10.1109/ICCCN.2016.7568580

[42] Ian J. Taylor and Andrew B. Harrison. 2009. Gnutella. In *From P2P and Grids to Services on the Web: Evolving Distributed Communities*, Ian J. Taylor and Andrew B. Harrison (Eds.). Springer London, London, 181–196. https://doi.org/10.1007/978-1-84800-123-7_10

[43] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. 2011. Invasive computing: An overview. *Multiprocessor system-on-chip: hardware design and tool integration* (2011), 241–268.

[44] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 1–4. https://doi.org/10.1109/MECO55406.2022.9797106

[45] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of VLDB Endow.* 14, 11 (jul 2021), 1964–1978. https://doi.org/10.14778/3476249.3476255

[46] 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

[47] 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

[48] 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 31 May 2024