Architecture of a Distributed Infrastructureless System

D. Mileski*+, M. Gusev*+
* Sts Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering
+ Innovation Dooel, Skopje, North Macedonia
E-mail: {dimitar.mileski, marjan.gushev}@finki.ukim.mk

***Abstract*—Recent trends in the production of portable devices, such as smartphones, smartwatches, and all other smart devices, show that their computing resources are comparable to laptops and desktop computers produced several years ago. In addition, many installed IoT devices around us realize that pervasive and ubiquitous computing integrates a remarkable amount of computing power. Unfortunately, these computing resources efficiently comply with the ever-increasing demand for computing power. Realizing a system that integrates devices on more minor architectural levels (IoT and edge layers) is a complex and challenging task, especially if the system goes beyond edge computing toward autonomous processing and realizes the essence of dew computing. The system should be platform-agnostic and provider-agnostic so all surrounding devices may participate and build more powerful computing resources. Finally, the proposed system realizes Infrastructureless computing, integrating Serverless, Deviceless, and Thingless computing as a service to nearby consumers. This paper analyzes the architecture of such a distributed computing system.**

***Keywords*—Dew Computing, Decentralized Supercomputer, Global Computing Power Market, Infrastructureless, Serverless, Deviceless, Thingless, Distributed Computing**

I. Introduction

A wide variety of devices incorporate computing and storage components, harnessed and offered as a service, ranging from powerful servers that often remain idle and underutilized to edge devices like smartphones, smartwatches, laptops, and credit card-sized computers such as Raspberry Pi [1], Orange Pi [2], Asus Thinker [3], and more. Various components within smart home systems possess computing and storage capabilities as services. The provider and the consumer are key stakeholders sharing IoT or smart devices, tablets, laptops, or similar computing resources in the edge computing environment.

The main idea is to provide computing resources, realizing the Infrastructureless computing concept [4], without worrying about resources and using the resources found nearby. This concept originates from serverless computing, where cloud providers dynamically manage the allocation of resources, generalizing the idea on lower architectural levels, including deviceless, which extends serverless computing on nearby devices rather than centralized servers [5]–[10] and Thingless computing [4]. In addition, users can leverage Deviceless as an alternative to the IoT-as-a-Service provided by the I/O cloud paradigm. Deviceless functions/actions operate on ephemeral and stateless containers, with a duration that may be limited to a single invocation [6]. Thingless further extends the concept of serverless computing by abstracting away physical dependencies, enabling interactions and computations without reliance on things (IoT).

We introduce a decentralized system formed in the proximity of the consumer as a distributed Infrastructureless system (DIS) to realize Deviceless and Thingless computing in various IoT and smart devices. This concept provides an environment where functions can run on nearby devices closer to the user, while Thingless computing extends this concept even on the ”things” architectural level [4]. The DIS approach enables a vast array of diverse devices to provide computing in a platform-agnostic way, integrating them into the cloud and enabling post-cloud edge computing and dew computing. In these scenarios, computing tasks do not solely execute on the cloud; instead, they involve function execution on diverse devices and things, expanding the scope beyond cloud-based execution.

Deviceless and Thingless computing can ensure scalable dew computing [11], where scalability occurs at the device level rather than the server level. The proposed DIS system can support scalable dew computing solutions [12].

The paper structure is as follows. Section II discusses the similarities and differences of the related work. Section III presents the system architecture and flow diagram for the computing providers and consumers. Implementation details are in Section IV, and Conclusion and Future Work in Section V.

II. Related Work

Mobile crowd computing includes device-to-device distributed computing [13], or smartphone Collaboration in data acquisition [14] with aspects different from our research. We focus on distributed and decentralized computing systems (decentralized supercomputers), providing examples of production systems and overviewing WebAssembly as an enabler for executing code at the Edge and Dew Computing in a platform-agnostic manner.

A. Distributed and Decentralized Computing Systems

The idea is to conduct radio SETI (Search for Extra-Terrestrial Intelligence) using a virtual supercomputer. SETI@Home [15] was officially launched in 1999 and

has remained operational for two decades, attracting over 5.2 million individuals worldwide until 2020 [16] [17].

The Golem Project [18] presents a decentralized supercomputer network offering an innovative approach to accessing computational power through a peer-to-peer network. This network allows users to rent computing resources from providers, making complex applications more cost-effective. The Golem ecosystem involves three key roles: providers supplying resources, requestors accessing those resources, and developers integrating software through the Application Registry. Golem envisions itself as a fundamental component of a decentralized internet but faces dependencies on other technologies. The Golem Network Token (GNT) is central to its operations, with security measures, including audits, ensuring resilience. Golem employs Docker containers to package, isolate, and execute computational tasks efficiently and securely within its decentralized supercomputer network.

The iExec project [19] builds upon existing research technologies on Desktop Grid computing to leverage underutilized internet-distributed computing resources, offering a cost-effective alternative for computationally intensive tasks. iExec's implementation uses XtremWeb-HEP, a mature open-source Desktop Grid software with essential features like fault tolerance, multi-application support, data management, and security. A notable research contribution within iExec's framework is the Proof-of-Contribution (PoCo) protocol, which introduces off-chain consensus. PoCo allows external resource providers to validate their contributions directly on the blockchain, enhancing transparency and accountability. iExec's research provides valuable insights and potential directions for further exploration in decentralized cloud computing.

SONM [20] is a decentralized fog supercomputer designed for general-purpose computing tasks, offering an alternative to centralized cloud services. SONM leverages IoT devices and implements fog computing, eliminating the need for expensive cloud infrastructure. It uses Ethereum Smart Contracts and its token, SNM, for resource access. SONM supports various applications, including scientific calculations, site hosting, game servers, neural networks, and rendering. SONM aims to reduce end-user costs by lowering bandwidth expenses, encouraging hardware contributions, and promoting competition. The architecture resembles a modular PC, comprising hub nodes (processors), miners (GPUs), a P2P message bus, clients, plugins, and a blockchain-based BIOS. The system utilizes the Slave Messaging Framework and intelligent contracts for network governance.

All these projects employ Docker [21] containers as a virtualization technology and do not address devices on a lower architectural level. While there is a consensus among development teams that Docker is a contemporary and advantageous solution, in this paper, we focus on using these concepts on the device and thing architectural levels and propose WebAssembly (WASM) [22] as an enabler for next generation Deviceless computing at the edge instead of Docker.

One significant difference between our proposed DIS solution and related work is using Deviceless and Thingless computing instead of Serverless computing, representing the most granular way of utilizing computing services and the most granular pricing model.

B. WebAssembly at the Edge

Ten out of 13 tests show that WebAssembly outperforms containerd container runtime using distro-less and distro-oriented container images comparing the cold start delays and total execution times of three WebAssembly runtimes: WasmEdge, Wasmer, and Wasmtime [23] confirming a promising enabler of next-generation serverless solutions.

The integration of WASM as a runtime environment for Serverless computing needs a specific focus on its application in edge computing [24], confronting the challenges posed by cold start latencies in traditional Docker-like Serverless frameworks. A new WebAssembly-based runtime environment (WOW) is integrated with Apache OpenWhisk to demonstrate significant reductions in cold start latencies (up to 99.5%), improvements in memory consumption (over five times), and substantial enhancements in function execution throughput (up to 4.2 times) on edge computing devices compared to Docker-based runtimes.

Evaluating three Serverless access patterns at the edge in scenarios from a campus camera network involves approximately 1,000 devices for real-time threat identification [25]. In a single client, multiple access workload, WASM is 1.5 to 3 times slower for moderate tasks and six times slower for basic tasks than OpenWhisk. Despite OpenWhisk's cold start penalties, it achieves native speed for subsequent calls, while WASM lags due to slower execution. In the Multiple Client, Single Access workload, WASM performs 70-90% faster on average than OpenWhisk, showing more stable response times. Combining both access patterns in Multiple Client, Multiple Access workload, OpenWhisk benefits from efficiency in Single Client, Multiple Access, offsetting the disadvantage from Multiple Client, Single Access. Wasm maintains consistent performance with latencies similar to previous workloads. Compared with containers, WASM's primary advantage is avoiding a significant cold start penalty with consistent performance and lower average latency than containers.

WASM implementation for Serverless computing at the Edge [26] overcomes inefficiencies in existing cloud-based solutions due to its lightweight and high-speed execution. The proposed aWsm framework, a native Wasm compiler, and runtime demonstrate promising startup latency of approximately $10\mu s$ to $30\mu s$ for null functions and a memory of 18KB to 179KB for benchmarks.

The aWsm functions cold-start latencies are smaller (0.2ms-0.4ms) than traditional VM/container cold-starts, indicating efficient Edge performance. Their approach includes sandboxing, customizable scheduling, and profiling

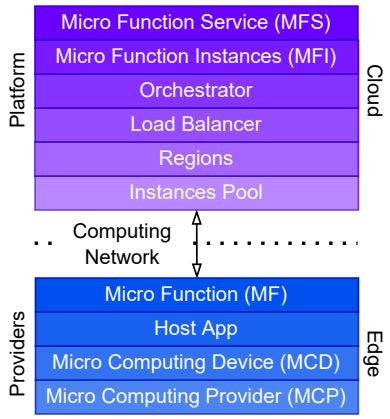

Fig. 1: Architecture of a Distributed Infrastructureless System

for efficient Serverless execution, addressing challenges associated with Edge computing.

"Infrastructureless computing" is a more general concept of providing computing infrastructure to nearby users without worrying about the resources, isolating programming from infrastructure specifications, and encompassing Serverless and Deviceless computing. In addition, it introduces a shift in computing layers, with functions executed on lower architectural levels, such as edge servers or devices. Examples of this model's potential include activating smartphones for overnight computing tasks or utilizing smart devices in parked cars [4], mainly to enhance the dew computing architectural model, making it a sophisticated platform for future architectural models. Our proposed DIS solution advocates a platform-agnostic operating system and computing access different from the serverless approach. WASM is implemented on devices and things instead of dockers on servers, particularly for a proof of concept. This approach aligns with the evolving landscape of decentralized cloud computing, contributing to the evolution of decentralized infrastructure and market networks.

III. DIS Architecture and Organization

This section specifies a DIS system describing the corresponding architecture, organization of the constituent parts, and flow activity diagrams.

A. Architecture

Fig. 1 presents the proposed DIS system's architecture in the cloud platform and edge providers' layer that consists of four integral parts (sublayers):

- *Micro Computing Provider (MCP)* that offers computing resources (processing, storage, and networking) as a service on the micro level;
- *Micro Computing Devices (MCD)* which implement MCP, including smartphones, laptops, PCs, or other devices with processing, storage, and networking capabilities, such that one or more MCDs can implement one MCP;

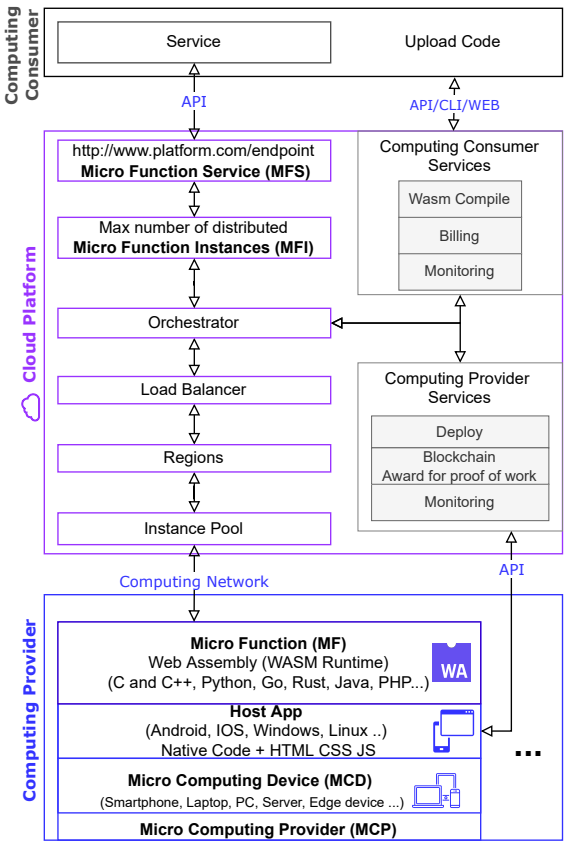

Fig. 2: Organization of a Distributed Infrastructureless System

- *Host Application* native to the device's platform (Android, iOS, Linux, Windows, etc.); and
- *Micro Functions (MF)* deployed on the host application.

The computing network serves as the medium for communication between the platform (cloud) and computing providers (edge). It connects a collection of providers offering devices with computing power for deploying MFs. Five internal sublayers constitute the cloud platform:

- *Instance Pool* as a set of MCDs to deploy MF and then call MF for execution;
- *Load Balancer* responsible for redirecting and balancing traffic among instances;
- *Orchestrator* to manage and orchestrate the rest of the microservices within the platform;
- *Micro Function Instances (MFI)* representing an abstraction over the Instance Pool; and
- *Micro Function Service (MFS)* provides a Function as a Service (FaaS) with elasticity and billing on a per-call basis, determined by the utilized vCPU and memory.

B. Organization

Fig. 2 presents a detailed DIS organization including:

- *Computing Provider (CP)* as an entity with Edge Devices offering computational tasks.

- *Computing Consumer (CC)* is an application or service needing computing resources, and
- *Cloud Platform* that empowers CC infrastructure, creates Micro Function Services (MFS), and runs these services on select Micro Computing Devices (MCDs).

The CP provides an MCD, a Smartphone, Laptop, PC, Server, or Edge device with computing, networking, and storage components. The Host App on the MCD executes code from Micro Functions (MFs), measures practical work, and monitors CP earnings.

Depending on the MCD's operating system, the Host App includes the native code, HTML, CSS, and JS necessary to run WebAssembly (WASM). The Host App also facilitates communication through APIs with Computing Provider Services in the platform. These cloud services encompass monitoring, deployment, and access to blockchain services that record MFS executions.

Each MF implements WASM code executed by a WASM Runtime. The Host App runs compiled WASM code (.wasm file) written in WASM programming languages [27] [28] and then compiled to WASM by Cloud Platform. When a CC writes the code for MFS and deploys it, the platform provides an endpoint that can be invoked, like the existing public cloud FaaS. After code deployment, the platform issues a URL (http://www.platform.com/endpoint) invoked when an external service, written by the CC, initiates communication with MFS.

The Orchestrator manages and coordinates calls to Micro Function Instances (MFI), Computing Consumer Services, Computing Provider Services, and the Load Balancer. The Load Balancer distributes traffic among instances.

A Region represents a group of instances in the same geographic area, defined by town, city, country, or even more granular geographic locations if the CP has Location Services enabled in the MCD. The Instance Pool is a collection of all available instances on the platform divided into Regions.

C. Flow Diagram

The flow diagram (Fig. 3) encompasses both the Micro Computing Provider (MCP) and the Computing Consumer (CC) to present request servicing. For example, if a user wants to offer computing resources in exchange for an award, the workflow consists of:

1) A new MCP user creates an account for the DIS platform.
2) The MCP user signs a contract to allow an awarding mechanism (proof of work defining the Unit of Work and the corresponding award).
3) The MCP user downloads a Host App (Android, iOS, Linux, Windows), which serves as the user interface for the DIS platform to monitor the execution status and total award. The Host App receives MFs to deploy and execute as the unit of work.

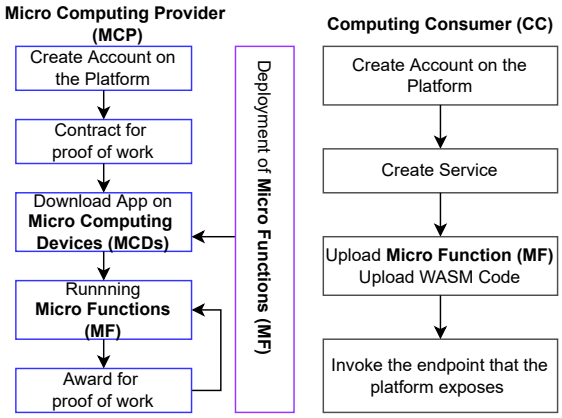

Fig. 3: Flow Diagram of a Distributed Infrastructureless System

Consumers manage their tasks through a command-line interface (CLI) or a graphical user interface (GUI). The interaction between MCPs and CCs within the system involves:

1) A new CC user creates an account.
2) The CC user creates a MFS.
3) The CC user uploads source code in WASM-supported programming language [27] [28].
4) DIS compiles the source code into a WASM module in the background.
5) The DIS platform deploys the WASM module to MCDs.
6) The CC user invokes a public endpoint URI exposed by DIS to execute the code.
7) The CC user receives the results of the execution and billing info.

IV. Implementations

A. Computing Provider

MCD requires an Internet connection, at least one installed operating system (Android, iOS, Windows, Linux, and MacOS), an installed Host App, and enabled location services. One approach to developing a Host App is to use multiplatform frameworks for different operating systems. Examples include Flutter [29], React Native [30], and Xamarin/.NET MAUI [31]. Another approach is to use a programming language that can be compiled for different operating systems and use a UI library. For instance, one of Go's (programming language) most powerful features is the ability to cross-build executables for any Go-supported foreign platform, making testing and package distribution much easier without access to a specific distribution platform. Applications written exclusively in Go can be constructed using the 'gomobile build' command [32], generating executable outputs for Android and iOS platforms.

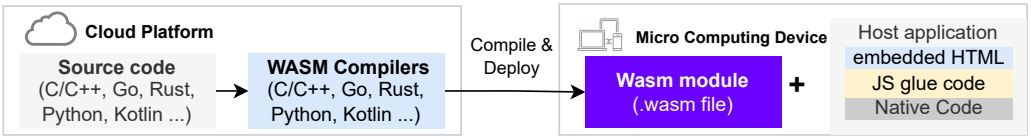

Fig. 4: Deployment of Distributed Infrastructureless System

The Fyne [33] UI toolkit can develop graphical user interfaces (UI) in Go across iOS, Android, Windows, and Linux, designed explicitly for creating native applications with native user interfaces.

Integrating WebAssembly with a Host App involves using WebAssembly APIs and often includes the creation of a JavaScript wrapper. While the WebAssembly API and JavaScript wrappers are standard, developers also use other tools, libraries, and approaches for this integration. WebAssembly runtimes [34], such as Wasmedge [35], optimized for edge computing with a focus on low memory footprint and quick startup times; Wasmtime [36], an open-source standalone runtime supporting both ahead-of-time (AOT) and just-in-time (JIT) compilation with increased execution flexibility; Wasmer [37], a general-purpose runtime designed for speed and lightweight deployment, suitable for various environments, such as desktop, server, and embedded systems.

MFs execute the .wasm file upon the request from the Cloud Platform and return a response (Fig. 2). The request triggers the execution of temporary stateless containers. Deviceless MFs run on ephemeral stateless containers (may only last for one invocation) [6].

B. Computing Consumer

Web, API, and CLI should be implemented as interfaces through which CC registers, logs in, uploads code, compiles, and executes. Various tools and technologies facilitate cross-platform development of interfaces using a unified codebase, including Go or NodeJS.

C. Cloud Platform

Communications between the Cloud Platform and the Host App on MCD can be realized as WebSocket tunnels with reverse tuning mechanisms to bypass firewalls and NATs. We deploy the containers with functions at the network edge (behind NATs and firewalls). The Computing Network can have WebSocket tunnels so Cloud Platform Services can use the remote containers and a new ID. Part of that Computing Network can include plain WebSocket control channels and REST APIs.

Another approach to implementing a Computing Network is to use Cloud Native Network [38]. Containers talk to each other and the infrastructure layer through a cloud-native network. Distributed applications have multiple components that use the network for different purposes. Tools in this category create a virtual network on top of existing networks specifically for apps to communicate, referred to as an overlay network. Some of the famous Cloud Native Networks are Cilium (graduated), Container Network Interface (CNI) (incubating), and FabEdge (sandbox) [38].

All Cloud Services can be implemented in Go (programming language) as a Cloud Native. Instance Pool is a Cloud Service with all instances divided into Regions, where a Region is a code abstraction of all instances in only a geographic area.

The Load Balancer can be Cloud Native Service Proxy [38] that functions as a tool that intercepts traffic going to or coming from a specific service, applies certain logic to it, and subsequently directs the traffic to another service. As an intermediary, it gathers information about network traffic and enforces rules. This role may range from a straightforward load balancer that directs traffic toward individual applications to a more intricate scenario involving a network of interconnected proxies running alongside individual containerized applications and managing all network connections.

Orchestration and scheduling refer to running and managing containers across a cluster, including Knative, wasmCloud, OpenFunction, or Kubernetes [38].

A stack-based virtual machine implements the platform-agnostic approach via WebAssembly as a binary instruction format. WASM is purposefully crafted as a versatile compilation target for programming languages, facilitating web deployment for client and server applications [22] and supporting JavaScript [39].

Fig. 4 presents MF deployment. A CC user uploads source code in one of the programming languages [27] [28], which is compiled by the DIS platform with CC Wasm Compile Service (Fig. 2) by supported WASM compilers, and the resulting WASM module is transferred to the MCD by CC and CP Deploy Service.

D. Non-functional requirements

1) Security and Privacy: Encryption protocols are vital for securing data transmission between CP and MCDs, ensuring privacy and data integrity.

2) Billing and Pricing: Possible billing and pricing models include a donation-based model without monetary transactions and a token-based system (cryptocurrency).

3) Energy and Battery Efficiency: Distributing computational tasks across a decentralized network of MCDs minimizes the impact on individual devices' battery life.

4) Monitoring and Debugging: DIS provides monitoring and debugging services for both MCP and CC.

5) Verification and Sandboxing capabilities: DIS prevents malicious actors from compromising platform integrity.

V. Conclusion and Future Work

DIS architecture realizes an Infrastructureless System distributed over many devices. Computing Providers can share their devices (smartphones, laptops, PCs, servers, edge devices, etc.) and offer their computing components with financial compensation. Numerous devices, often underutilized computing power, can be leased out. This architecture also facilitates collaboration among computing consumers (developers, scientists, etc.) who can execute tasks using these services, leveraging serverless features and utilizing devices provided by computing providers. A central component of the architecture presented in the paper is WebAssembly (WASM), serving as a crucial factor for platform-agnostic processing in advancing toward the next generation of Thingless, Deviceless, and Serverless computing, collectively referred to as Infrastructureless computing.

Implementation is a crucial focus for future work, exploring how potentially thousands or millions of devices can effectively address the challenges of parallel and distributed processing. Open challenges include integration in Massively Parallel Processing and how this system can serve as a foundational platform for developing services in the cloud continuum (dew, edge, and fog).

References

[1] E. Upton and G. Halfacree, *Raspberry Pi user guide*. John Wiley & Sons, 2016.

[2] T. S. Love, J. Tomlinson, and D. Dunn, "The orange pi: Integrating programming through electronic technology," *Technology and Engineering Teacher*, vol. 76, no. 2, p. 24, 2016.

[3] L. Clark and L. Clark, "What is the asus tinker board?" *Practical Tinker Board: Getting Started and Building Projects with the ASUS Single-Board Computer*, pp. 3–11, 2019.

[4] M. Gusev, "Serverless and deviceless dew computing: Founding an infrastructureless computing," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1814–1818.

[5] G. Merlino, G. Tricomi, L. D'Agati, Z. Benomar, F. Longo, and A. Puliafito, "Faas for iot: Evolving serverless towards deviceless in i/oclouds," *Future Generation Computer Systems*, 2024.

[6] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "Deviceless: A serverless approach for the internet of things," in *2021 ITU Kaleidoscope: Connecting Physical and Virtual Worlds (ITU K)*. IEEE, 2021, pp. 1–8.

[7] M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Dustdar, O. Scekic, T. Rausch, S. Nastic, S. Ristov, and T. Fahringer, "A deviceless edge computing approach for streaming iot applications," *IEEE Internet Computing*, vol. 23, no. 1, pp. 37–45, 2019.

[8] N. Vance, M. T. Rashid, D. Zhang, and D. Wang, "Towards reliability in online high-churn edge computing: A deviceless pipelining approach," in *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2019, pp. 301–308.

[9] S. Nastic and S. Dustdar, "Towards deviceless edge computing: Challenges, design aspects, and models for serverless paradigm at the edge," *The Essence of Software Engineering*, pp. 121–136, 2018.

[10] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: extending serverless computing to the edge of the network," in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–1.

[11] M. Gusev, "Scalable dew computing," *Applied Sciences*, vol. 12, no. 19, p. 9510, 2022.

[12] D. Mileski, M. Gusev, and D. Spasov, "A promising foundation for scalable dew computing towards deviceless and thingless computing," in *2023 ICT Innovations, web proceedings*, 2023, pp. 1–10.

[13] D. Remédios, A. Teófilo, H. Paulino, and J. Lourenço, "Mobile device-to-device distributed computing using data sets," in *proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services on 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2015, pp. 297–298.

[14] L. Duan, T. Kubo, K. Sugiyama, J. Huang, T. Hasegawa, and J. Walrand, "Motivating smartphone collaboration in data acquisition and distributed computing," *IEEE Transactions on Mobile Computing*, vol. 13, no. 10, pp. 2320–2333, 2014.

[15] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[16] "SETI@home hibernation," https://setiathome.berkeley.edu/forum_thread.php?id=85267, [Accessed 12-10-2023].

[17] "SETI@home," https://setiathome.berkeley.edu/, [Accessed 12-10-2023].

[18] Golem, [Accessed on 20.04.2023]. [Online]. Available: https://assets.website-files.com/60005e3965a10f31d245af87/60352707e6dd742743c75764_Golemwhitepaper.pdf

[19] "Iex.ec," [Accessed on 20.04.2023]. [Online]. Available: https://iex.ec/wp-content/uploads/2022/09/iexec_whitepaper.pdf

[20] SONM, [Accessed on 20.04.2023]. [Online]. Available: https://www.allcryptowhitepapers.com/wp-content/uploads/2018/05/SONM-Whitepaper.pdf

[21] "Overview of the get started guide - docs.docker.com," https://docs.docker.com/get-started/, [Accessed 12-10-2023].

[22] "WebAssembly," https://webassembly.org/, [Accessed 12-10-2023].

[23] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *Journal of Grid Computing*, vol. 21, no. 3, p. 34, 2023.

[24] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[25] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[26] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 261–2615.

[27] "Awesome-wasm GitHub," https://github.com/mbasso/awesome-wasm, [Accessed 24-01-2024].

[28] "awesome-wasm-langs GitHub," https://github.com/appcypher/awesome-wasm-langs, [Accessed 24-01-2024].

[29] "Flutter documentation," https://docs.flutter.dev/, [Accessed 22-01-2024].

[30] "React native documentation," https://reactnative.dev/docs/getting-started, [Accessed 22-01-2024].

[31] ".NET MAUI documentation," https://dotnet.microsoft.com/en-us/apps/maui, [Accessed 22-01-2024].

[32] "Gomobile command documentation," https://pkg.go.dev/golang.org/x/mobile/cmd/gomobile, [Accessed 22-01-2024].

[33] "Fyne," https://fyne.io/, [Accessed 24-01-2024].

[34] "CNCF Landscape wasm," https://landscape.cncf.io/?group=wasm&view-mode=grid, [Accessed 23-01-2024].

[35] "Wasmedge documentation," https://wasmedge.org/docs/, [Accessed 22-01-2024].

[36] "Wasmtime documentation," https://docs.wasmtime.dev/, [Accessed 22-01-2024].

[37] "Wasmer documentation," https://docs.wasmer.io/, [Accessed 22-01-2024].

[38] "CNCF Landscape," https://landscape.cncf.io/, [Accessed 22-01-2024].

[39] "WebAssembly Concepts - WebAssembly — MDN," https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts, [Accessed 12-10-2023].