future internet

MDPI

Article

Multi-Agent Dynamic Fog Service Placement Approach

Nerijus Šatkauskas and Algimantas Venčkauskas *

Faculty of Informatics, Kaunas University of Technology, Kaunas LT-51368, Lithuania; nerijus.satkauskas@ktu.edu
* Correspondence: algimantas.venckauskas@ktu.lt

Abstract: Fog computing as a paradigm was offered more than a decade ago to solve Cloud Computing issues. Long transmission distances, higher data flow, data loss, latency, and energy consumption lead to providing services at the edge of the network. But, fog devices are known for being mobile and heterogenous. Their resources can be limited, and their availability can be constantly changing. A service placement optimization is needed to meet the QoS requirements. We propose a service placement orchestration, which functions as a multi-agent system. Fog computing services are represented by agents that can both work independently and cooperate. Service placement is being completed by a two-stage optimization method. Our service placement orchestrator is distributed, services are discovered dynamically, resources can be monitored, and communication messages among fog nodes can be signed and encrypted as a solution to the weakness of multi-agent systems due to the lack of monitoring tools and security.

Keywords: fog; fog computing; service placement; placement; fog service placement; multi-agent systems

1. Introduction

IoT infrastructure is shaping the world in the way that the exchange of data between physical world items and the virtual world has become more and more common. It is said that the number of IoT devices connected to the network might be currently around 50 billion according to the source [1]. The amount of their generated data could reach 79.4 zettabytes by 2025. It can be considered as a global network of an infrastructure with numerous multiple devices, which are made up of different parts including sensing, communication, networking, and information processing [2]. However, long data transmission distances between the end-users and cloud servers lead to a higher network data flow, data loss, latency, and energy usage [3]. This is where fog computing emerges with an intention to bring computation and storage capabilities to the edge of the network. A distributed fog computing infrastructure is effective, with delay-sensitive IoT applications rendering minimal latency and energy consumption to process data while using resource-limited fog/edge devices.

Fog computing has typically a layered architecture with three different layers: end-device (terminal) layer, fog layer, and cloud layer [4]. An end-device layer consists of end-devices that are distributed around the accessible area. They are designed to collect data in order to transmit them to upper layers. A fog layer, meanwhile, is situated at the edge of a network between a fog layer and an end-device layer. It is designed for computing, filtering, joining, transforming [5], and the storage of data. It can be both mobile and static. A cloud layer consists of storage devices and servers with high performance. However, the number of layers or their names in the architecture may slightly differ like in [6], where an orchestrator has its own layer between a fog layer and an IoT application layer to make four layers in total. This paper [7] suggests using two fog layers instead where the first one consists of small to medium calculation capacity nodes, and the second one is made of powerful ones. As for the fog radio access networks (F-RANs) to support 5G, there is a network access layer between a cloud layer and a logical fog layer [8]. And five layers are identified in the reference architecture [9]: (1) sensors, edge devices, and gateways,

check for updates

Citation: Šatkauskas, N.; Venčkauskas, A. Multi-Agent Dynamic Fog Service Placement Approach. *Future Internet* **2024**, *16*, 248. https://doi.org/10.3390/fi16070248

Academic Editors: Stefano Rinaldi and Alan Oliveira De Sá

Received: 14 June 2024
Revised: 9 July 2024
Accepted: 11 July 2024
Published: 13 July 2024

(2) network, (3) cloud services and resources, (4) software-defined resource management, and (5) IoT applications and solutions. Please see Figure 1 for more details.

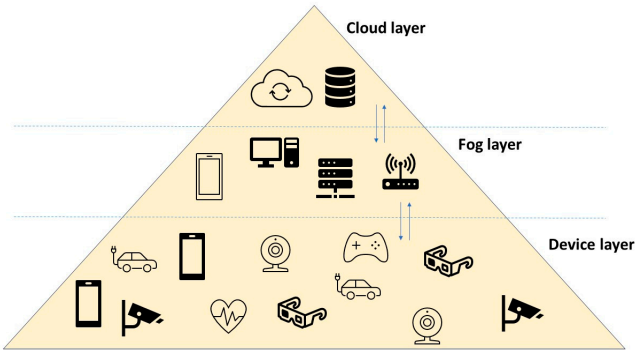

Figure 1. Typical fog computing architecture.

Even though the number of fog computing layers may sometimes slightly differ, the end-device layer will always consist of IoT devices such as smartphones, tables, computers, sensors [10], remote machines, and smart vehicles [11], which communicate in a wired or wireless manner [12] with the fog layer. These end-devices may have limited resources [13] such as storage and computational power, as well as limited bandwidth.

Multi-agent systems (MASs) in turn, as the review article [14] claims, gained excessive attention from scientists of different areas such as a computer science and civil engineering. They can solve complex tasks by breaking them down into smaller ones. Each task can be assigned to agents that function independently from each other. Agents can make decisions on taking actions based on their action history, interactions, and its goal. Multi-agent systems recently have been a popular research topic, and they are applied in such areas as unmanned aerial vehicles, industrial internet of things, and wireless sensor networks as per the publication [15]. However, as it added in the same publication, irrespective of all the benefits, multi-agent systems are very vulnerable to network attacks due to an open communication environment and the complexity of the system. It lacks an integrated system to monitor and manage the activities of all the network nodes. Information exchange is usually very high, but the information flow cannot be verified, and therefore the system is at a security risk.

This paper includes the following sections: Introduction, Related Work Review, Service Placement Orchestrator Implementation, Materials and Methods, Results, Discussion, and Conclusions. Apart from general information in the Introduction, the Related Work Review section gives more focused details of the problems and potential solutions in other research works. Orchestrator design is defined in the Service Placement Orchestrator Implementation section based on relevant characteristics. The chapter of Materials and Methods discusses the hardware and software as well as test methods. The Results, Discussion, and Conclusions sections give an insight into the experimental results, followed by assumptions and result conclusions.

2. Related Work Review

Computing is still a developing paradigm and it has plenty of challenges to overcome. A review publication [1] identified a number of such challenges, but some of them include mobility, scalability, availability and reliability, resource management, application placement strategies, and security and privacy. Its heterogeneous nature may lead to structural challenges. Some fog nodes may have limited resources; therefore, there is a need to develop distributed applications [4]. Additionally, it can be difficult to maintain service access authentication to preserve privacy. End-devices are closer to their users; therefore, they can collect more sensitive data. It raises privacy concerns for end-users. Private location and personal data can be disclosed by an untrusted party hacking a poorly protected node [16].

Applications usually have some objectives and constraints [17]. A set of objectives can be conveyed as diverse QoS requirements. Constraints, in their turn, can be application-related or network-related. It may include meeting the deadline of applications, bandwidth requirements, security, privacy, power consumption etc. Fog devices are highly distributed and resource constrained [18], which is opposite to Cloud Computing. One of the key problems to run an application in the fog environment is resource allocation. The purpose of this is to select devices that have available resources for the required application services. Resource allocation can be mainly divided into three categories: resource placement, resource scheduling, and resource migration. To be more specific, resource placement defines where to place resources, resource scheduling defines the time when it has to happen and the scale of resources, meanwhile resource migration determines where these resources can be moved. While doing this, resource monitoring and metrics have to be considered.

Service placement has to be as optimized as much as it is possible for fog nodes to use their resources efficiently [17]. The main purpose of an optimization is to reduce or increase certain features based on the objectives [19]. The optimization itself by its nature can be (a) heuristic, (b) metaheuristic, (c) machine learning, (d) mathematical programming, and diverse [17]. A heuristic approach offers a sub-optimal solution with the consideration of time. A metaheuristic approach provides an optimal solution with methods that are nature inspired and are not focused on a local optimum. Machine learning is overly dependent on network resources and the quality of training, meanwhile mathematical programming is meant for a single performance parameter optimization. The PSO algorithm as a metaheuristic method seems to be well suited to a multi-objective optimization, except that it can lead to a premature convergence and a local optimum when a diversity of the population is insufficient [20], which may require some extra attention.

Optimization can be based on a single-objective problem and a multi-objective problem. A multi-objective optimization can lead to trade-offs to fulfill conflicting goals [21] simultaneously, while taking constraints into consideration [22]. There is no single best solution. It gives a set of solutions. And, there is a large number of centralized optimization techniques as the paper [23] suggests. However, centralized solutions require a centralized controller to keep track of the global system information. But the problem with a centralized solution is that if a controller node fails, the whole system fails. It creates a single point of failure [24]. Decentralized or a distributed design can be an answer to his issue since multiple fog controllers are involved in making a placement decision. The downside of this approach is that it takes additional efforts to identify the fog nodes fit enough to take a controller node role. And, service placement task scheduling algorithms can be divided into immediate, batch, preemptive, non-preemptive, static, and dynamic [25]. What dynamic service placement algorithms set apart is that service discovery mechanisms are defined in such a way that available services are known to other services due to their interaction [26]. Each of the service (microservice) enquiries are dynamically updated in the service registry to determine availabilities.

A service placement solving technique can also be offline and online [27]. In an offline technique, all the requirements and constraints are known in advance. To be more precise, it can be said that a placement decision is made in the compile time [28]. Meanwhile, online placement decisions are made in the runtime. It is, however, more beneficial to consider a placement as an online technique. It is more dynamic and capable of reacting to current changes in the infrastructure.

Dynamic networks are the ones where both the mobile fog nodes and the end-users change their characteristics within the time, including network topology changes [29]. A fog computing infrastructure has to be mobility aware, but it is challenging due to its dynamics. Dynamicity can be related to the fog infrastructure dynamicity and to the application dynamicity [28]. Fog computing architecture is highly dynamic [20] and its nodes may join or leave the network any time. But the resources in that particular node may not be sufficient to host a requested service that leads to a lower QoS, longer response time,

or even service failure. Applications, therefore, should be deployed/removed dynamically while considering the capabilities of the changing fog infrastructure [28].

Please see Table 1 for a review summary.

Table 1. Related work summary.

Ref.	Solution	Dynamic Placement	Distributed	Resilience
[30]	Application Module placement algorithm	✓	-	-
[31]	ECC platform	✓	-	-
[32]	Management modules	✓	-	-
[33]	PSO-based metaheuristic and a greedy heuristic algorithm	✓	-	-
[34]	Decentralized algorithm	✓	-	-
[35]	Folo: Dynamic task allocation framework	✓	-	-
[36]	Decentralized replica placement	✓	✓	-
[37]	Optimization framework	✓	✓	-
[38]	MM and RM framework	✓	✓ (both)	✓
[39]	ELECTRE load balancing algorithm	✓	-	✓
[40]	MicroFog framework	-	✓	-
[41]	S-HIDRA architecture	-	✓	-
[27]	A3C algorithm	✓	✓	-
[42]	Kubernetes framework	✓	-	-
[43]	Framework	✓	-	-
[44]	Two-stage optimization model	✓	-	✓
[45]	DCSP method	✓	✓	-
[46]	ANFIS and GAO	-	-	-
[47]	CFS model	✓	-	-
[48]	CBR-MADE-k model	✓	✓	-
[49]	Orchestration and management solution	✓	✓	-
[50]	MILP model	-	-	✓

There are numerous research papers trying to solve a service placement or an application placement using a dynamic approach. Some of them use a distributed control method instead of a centralized or federated one. Some of them focus on resilience. But there are almost none of them that are dedicated to a dynamic service placement in a distributed way with an intention to keep it more resilient due to a distributed approach.

3. Service Placement Orchestrator Implementation

3.1. Design Motivation

The aim of this orchestrator design process is to demonstrate how orchestrators make decisions to control their services in respect to the QoS and security requirements. Each decision the orchestrators make to start/stop/move their services must be verified in a dynamic way, whether the minimal requirements related to various hardware and software restrictions of the involved hardware devices, as well as requirements due to peculiarities of the application area (e.g., sensitive data should be protected better than environment monitoring data), are met.

The orchestrator design should consider its three main stages:

- Each orchestrator as a part of the first stage should take into account such requirements as security, CPU, RAM, and power based on the application area and diverse fog node hardware/software capabilities to decide if it is possible to launch all required services without violating these requirements;
- The second stage is to find an optimal distribution for deployable services among different fog nodes. It can be vital for saving energy and computation resources in cases when some services need to be stopped, suspended, or moved to other fog nodes;

- The third stage is a dynamic service placement for a situation when circumstances change during the runtime, and orchestrators need to change the distribution of their services among available fog nodes according to these new conditions.

The orchestrator in the first stage should have all the information of services and placements in all the fog nodes collected and synchronized, and should be aware of the available QoS and security requirements. If all minimal requirements are satisfied, services can be launched. During the second stage, an orchestrator should search for an optimal placement of its services. The orchestrator should detect in the third stage certain changes in its resources or environment indicators, process this data, and relocate its services from the current fog node to another one.

Having considered design requirements and knowing the benefits of multi-agent systems, it provides a good starting point. MAS behaves like a network that can correct itself and analyze itself [51]. These intelligent network nodes can both function individually and cooperate to pursue their general and individual goals. The integration of a MAS constitutes a complex framework designed for system control and optimization. However, security issues must be addressed to maintain its resilience, and resource monitoring needs a solution due to the lack of an integrated tool.

Particle Swarm Optimization (PSO) is a population-based metaheuristic technique that is used to solve optimization problems [52]. It imitates a social behavior of birds where each bird in the flock, based on its individual experience and social experience, approaches their target food. It is a principle of social interaction to solve the problem. This technique is good to optimize continues non-linear functions. As in the wild with a flock of birds, here PSO starts with a swarm of potential solutions. Each potential solution is represented by a particle. The population with each iteration is updated by updating the particle's velocity and position. These updates are based on the personal best value and global best value. Each particle converges to its new position until the global optimum is found. Multiple objectives, however, require IMOPSO for a set of non-dominated service placements.

The Analytical Hierarchy Process (AHP) is used as a second technique to choose the best solution from a Pareto set. AHP, which was developed by Saaty, is a technique that helps to simplify complex and poorly structured problems by making a number of pairwise comparisons [53]. Decision criteria are organized in a hierarchical way, and they are given their weight coefficients based on a potential impact to achieve the desired goal. At the end of the process, the best service distribution alternative is chosen, which corresponds to the highest criteria priority as the final optimization process output.

In order to meet design requirements, the whole orchestrator design process is broken down into separate subsections, which include resource monitoring, starting new services, data synchronization, security maintenance, and the orchestrator architecture itself. All these subsections are needed to design a service placement orchestrator as a multi-agent system that overcomes its inherent shortcomings of network attack vulnerabilities and the absence of an integrated monitoring tool.

3.2. Resource Monitoring

Resource monitoring is implemented using a monitoring agent. It uses a cyclic behavior to wait for messages. The content of these messages is filtered with the method startsWith() in order to take a relevant action. It can obtain messages from a battery voltage agent (BattVoltAgent), light agent (LightingAgent), or a sensor agent (AbstractSensorAgent). A battery voltage agent keeps track of Raspberry Pi 4 battery charge level as a fog node. Pi 4 does not have a native analogue-to-digital (ADC) conversion option. An external one such as the MCP3424 18-Bit ADC-4 channel converter would be needed. However, a high or low pin 3 voltage is checked for simulation purposes using the Pi4J library.

A sensor agent is used to monitor such external resources as light intensiveness or temperature etc. A 5 s interval is used as a sensor update interval, which involves communication between end-devices and particular agents and as a sensor agent poll interval, which involves communication between agents and the orchestrator. There are

a few commands that a sensor agent can receive like Get, Set, Move, or Changeip. If a sensor agent receives a Get message, it identifies a sender and responds with a value that is obtained using the getValue() method as an ACL INFORM message. These messages are sent as a part of a regular sensor polling task at a predefined interval like 5 s. It can be also triggered once a threshold value is reached.

Communication with the end-devices to obtain their values is completed using the COAP protocol. Lighting and temperature agents keep on polling their end-devices for their values as a part of an onTick() event, which is periodically triggered after a timeout interval. The COAP request method GET is used to obtain a value from the end-point device. The PUT method is used to set a value instead.

As in Figure 2 shown above, an end-device keeps on periodically communicating with its fog node. Once a monitoring agent detects that its battery voltage is below a required level, it sends a service redistribution request to the fog node Decision Maker agent. A remote service redeployment is calculated using IMOPSO and AHP, and its solution is communicated to the Execution agent. Services in the current fog node are stopped. A redeployment solution is synchronized by the current fog node and the fog node where the services have to be moved to. A Synchronization agent sends a request to its Execution agent and the end-device service is assigned to the fog node 1.

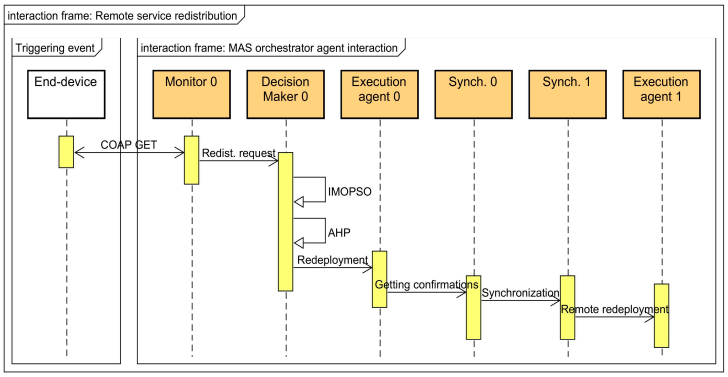

Figure 2. Remote service redeployment.

3.3. Starting New Service

Service requests are generated when a new end-device appears, or the current end-device is moving from one place to another. End-devices at the current fog node are disconnected and they need to send a request to a closer fog node. These end-devices are identified by their end-point address such as *coap://192.168.0.24/temp* for a temperature device or *coap://192.168.0.24/led* for lights, which can be changed or adjusted as required. Please see Figure 3 for more details.

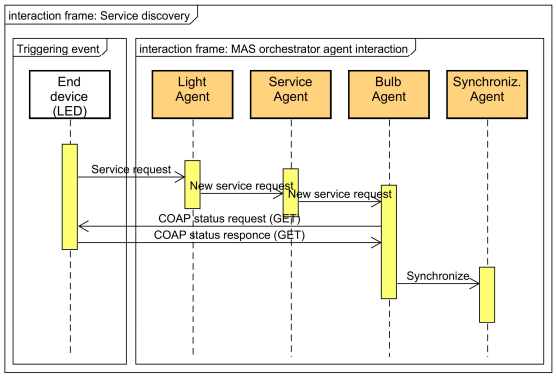

Figure 3. Starting a new service.

The light agent works as a light sensing element. If the light level outside falls below a certain threshold, it sends a request to its Service agent to start a new service. The Service agent defines what that service should be like. When that is a light service, it can define what intensity of the light is needed. The Service agent afterwards sends a defined request to the Bulb agent. It uses the PUT method to communicate with the end-device to set a required level of light. As the level is adjusted, the outcome is synchronized.

3.4. Data Synchronization

Currently, synchronization among orchestrator agents and communication among fog nodes is implemented as a three-way communication topology. Upwards communication is bound for a vertical synchronization with a parent agent by going one level up within the hierarchy. Downwards communication is meant for sending messages down by one level to a child agent. This is completed by sending ACL INFORM messages within internal fog node agents. Sideways communication is horizontal communication among nearby fog nodes, and it is being completed by Synchronization agents. All the fog nodes need to keep their data about resources and statuses updated to make their informed decisions for the best possible service placement when starting new services. This is also necessary when currently available services are getting relocated due to resource or security restrictions. All the horizontal communication now is being completed via Wi-Fi, but it can also be completed via Bluetooth, ZigBee, or even Ethernet. Please see Figure 4 for more details.

```
 :sender  ( agent-identifier :name FogOrchestrator4
@fog4  :addresses (sequence http://192.168.0.90:7774 ))
 :receiver  (set ( agent-identifier :name FogOrchestrator4
@fog4  :addresses (sequence http://192.168.0.90:7774 )) )
 :content  "AgentList:4:9:FogOrchestrator4:sniffer0:sniffer0-
on-fog4-cnt:Temp4:sniffer1:Light4:Decision4:Bulb4:sniffer1-
on-fog4-cnt"
```

Figure 4. Agent list synchronization messages.

To keep an agent list synchronized, it is updated using the AMS Service component. This component allows us to launch a search based on certain constraints and descriptions. It monitors agent registration, deregistration, and tracks them. A GetAgentList message is sent to all known orchestrators in the fog nodes to retrieve a list. Meanwhile, there is a list of possible orchestrators with their IP addresses stored, but their presence in the fog network is confirmed with a response. Please see Figure 5 for more details.

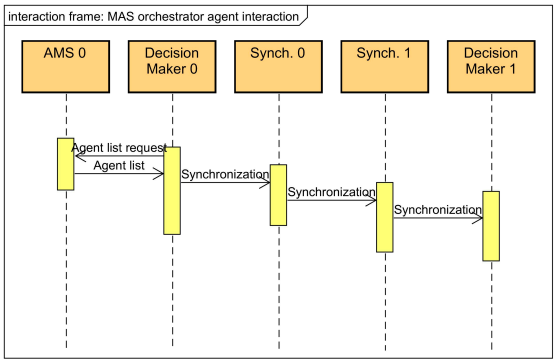

Figure 5. Agent list synchronization.

3.5. Security Maintenance

As a default configuration, agent communication messages are neither encrypted nor signed. It may give an opportunity for a hacker using a malicious agent to sniff a message content or even modify it. It would lead to malicious instructions for a recipient agent. Different requests can be sent to AMS to eliminate some agents if there are no security checks. To maintain security, the JADE security add-on JADE-S was used. After

it is installed, services such as SecurityService, PermissionService, SignatureService, and EncryptionService have to be enabled. SecurityService is a primary one, and the other ones are optional and can be enabled as required.

Instead of using the method send(), the method sendMessage() is used. It allows us to specify whether the message has to be signed and encrypted. Credentials are checked by the method retrievePrincipal(). It is triggered first before a relevant message is sent. It first sends a *request* message with the content "get-principal", and the answer is formed by a recipient to send it back as an *inform* message. Please see Figure 6 for more details.

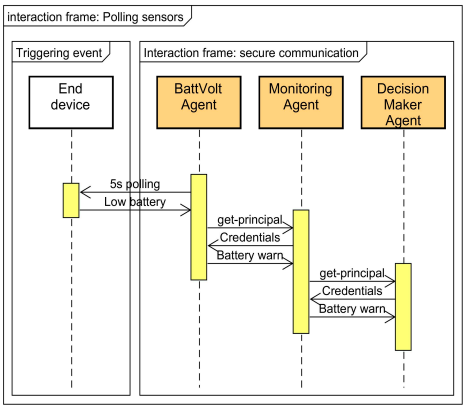

Figure 6. Retrieving credentials in secure communication.

In addition to the above information, further modules have to be enabled to adjust default security settings to preferred ones. The SignAlgorithm allows us to choose an algorithm that will be used to have the messages signed such as SHA1withRSA, MD5withRSA, DSA etc. The size for public and private keys can be defined by the module AsymKeySize, which ranges from 512 as a default value to 2048. There are a few more modules, in addition to the above ones, used to ensure additional customizability.

3.6. Dynamic Orchestrator Architecture

A service placement orchestrator was implemented as a multi-agent system (MAS) Java application using a JADE framework. It allows us to develop MAS applications that comply with FIPA specifications. It offers such features as an agent abstraction, asynchronous messaging, and a service discovery based on the yellow pages method. The orchestrator is implemented as a distributed monitoring, decision making, and execution model, which is available in each fog node within a relevant fog computing system. Continuous resource monitoring and request processing allow us to make informed decisions in a dynamic way. Service distribution among multiple nodes contributes to the enhancement of computational output, power usage, and resilience. A distributed orchestrator is more resilient than a centralized one because of the absence of a single point of failure (SPF). Mobility, changing resource levels, and fog node failures preferably lead to dynamic and distributed decision making.

Once the fog nodes connect to the same network to form a shared infrastructure, they start monitoring their resources such as CPU, RAM, battery, and security. The Monitoring agent waits for resource-related messages using a cyclic behavior. These messages are classified by use cases and tagged with relevant resource levels. A Request agent is waiting for service request messages. They are also classified by use cases. Due to a security add-on, JADE-S can be either signed, encrypted, or both signed and encrypted. Required security measures and used security measures are identified among involved agents and the messages can be discarded if these requirements are not met. Please see Figure 7 for more architecture details.

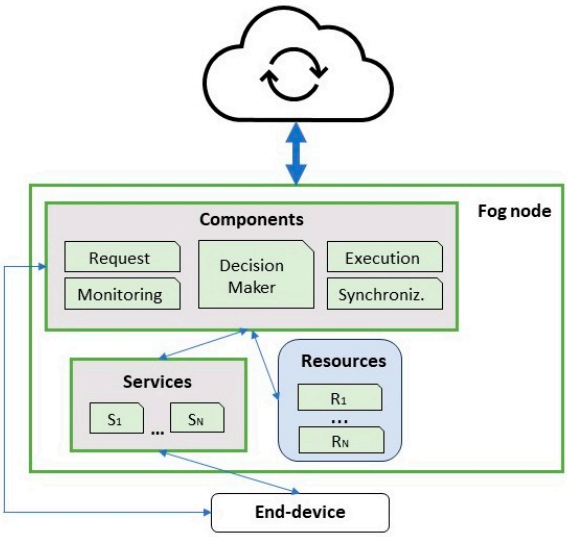

Figure 7. Proposed distributed orchestrator architecture.

The Decision Maker agent can obtain messages either from the Request agent, Resource Monitoring agent, or from the Synchronization agent. Messages from a Request agent can be related to a new service request or a current service redeployment. The Monitoring agent keeps its Decision Maker informed if resources are below a required level. There is also a Synchronization agent that keeps resource data synchronized among the involved fog nodes to let a Decision Maker make informed decisions when choosing a local or a remote service distribution. The Decision Maker uses IMOPSO and AHP algorithms to calculate the best deployment based on the objective functions. IMOPSO is used as a primary request processing algorithm for a higher number of options. Once these options are considered by the Decision Maker, a potential solution is further processed with a reference to objective functions. The AHP algorithm uses its criteria matrix to make a final placement decision based on prioritized criteria. Four criteria are used at the moment, but this number can be adjusted as required.

As soon as a placement decision is made, it is communicated to the Execution agent. It runs a cyclic behavior waiting for its messages. Received messages are classified based on use cases or key words such as "Decision" for internal purposes. If there is a local redeployment within the same JADE platform, agents can be moved from one container to another. Agents can not be moved, however, to another platform. When services have to be redeployed remotely, current agents are killed and the other ones with the same parameters are created in a remote platform. Any changes in the fog computing infrastructure, whether they involve a new service placement or a remote redistribution, are synchronized by a Synchronization agent. Additional details about the architecture are available in the publication [54].

3.7. Service Placement Decision-Making Method

The method that was used for a service placement in this research was proposed in [55], and it is made of two stages. The first stage uses IMOPSO and the second one uses AHP. IMOPSO is a slightly adapted version of MOPSO, which was originally introduced by Coello et. all in [56]. AHP was first introduced R. W. Saaty in [57]. The purpose of this two-stage method is to distribute n services among k fog nodes. All the QoS features of the i-th placement for X_i are defined by objective functions $f_j(x)$, $j = 1, 2, \ldots, m$. An optimization process seeks to find the best service placement X_{opt} while minimizing its objective functions f_j:

$$X_{opt} = \underset{i}{\operatorname{argmin}} F(X_i). \quad (1)$$

As a part of fog computing paradigm, fog nodes can have different technical capabilities, network throughput, and security units. Such qualities to consider may include CPU, RAM, power usage, network range, communication security protocols and authentication etc. Optimization based on one feature may come at the expense of neglecting other ones. Therefore, there is no universal solution to a multi-objective task. This leads to obtaining a set of non-dominated solutions based on fog node constraints. Please see Figure 8 below for more details.

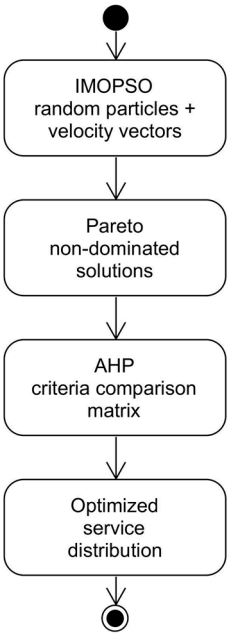

Figure 8. Optimal service placement method flow chart.

The IMOPSO method is used to obtain Pareto potential solutions. Since these placements, which are added to a repository, are non-dominated, it means that they are better by some criteria scores. AHP is used for the final best placement based on prioritized criteria. Alternatives are compared with each other using a judgement matrix. More details about an implementation of the IMOPSO algorithm and AHP process are provided in the publication [55]. A short summary is given below.

The IMOPSO algorithm is applied in the following way as it is in Figure 9:

1. A swarm is generated based on predefined test parameters such as a number of objective functions, particles, epochs, inertia weight, cognitive coefficient, social coefficient, number or services, and a range of particles. Particle positions with the swarm are randomly assigned. Initial values such as a global best score and position as well as velocities are given;
2. Each particle position gets evaluated for its new individual score based on its objective function res = $(x - y)^2$;
3. Velocity is updated using the formula oldVelocity[i] = inertia + (pBest[i] − pos[i]) × cognitiveComponent × r1 + (gBest[i] − pos[i]) × socialComponent × r2;
4. Individual particle positions are updated by adding its velocity to its position;
5. Searching for individual best scores and individual best particle positions within the swarm;
6. Searching for the global best score and global best position;
7. If a particle dominates or if it neither dominates nor is dominated, it is added to a repository as an individual best result or global best result, respectively;
8. Repeat until the required number of cycles is completed.

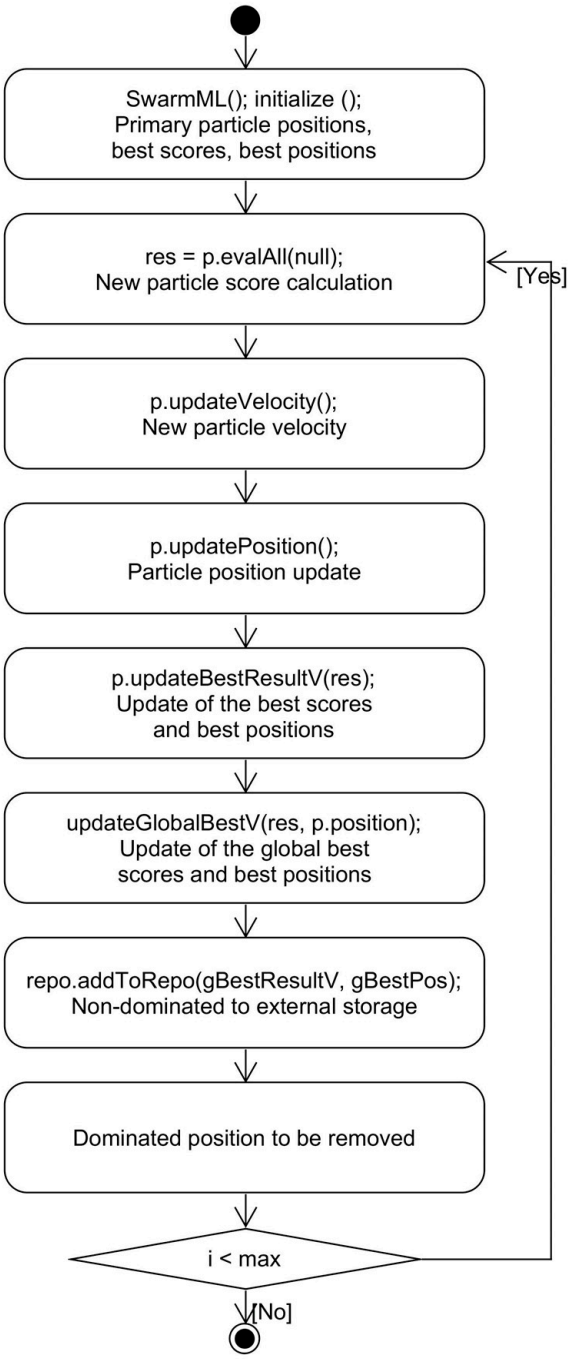

Figure 9. IMOPSO algorithm.

The AHP algorithm is applied in the following way as it is in Figure 10:

1. Repository particle positions in prepo are pairwise compared. This is completed with each criterion (objective function) fs[i];
2. Comparison results are stored in the array allComp[i];
3. The priority vector eigenvectorC is to be calculated. It is a matrix normalized Eigen vector. Each column of the priority matrix Ccomp is summed up. Each element of the column is divided by its sum to obtain a normalized relative weight in the double array matrixnormC;
4. The transposed matrix eigenFinal is used to find the best particle position. The best service distribution is the alternative that has the highest level of priority.

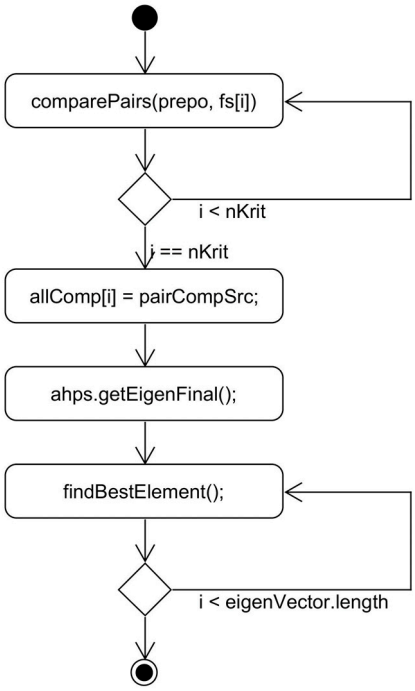

Figure 10. AHP algorithm.

4. Materials and Methods

4.1. Experimental Tools

A Raspberry Pi 4 Model B computer with 4 GB of RAM was used as a low-resource fog node. It runs Raspbian 10 OS as its operating system and JDK 1.8.0_333 as a Java Development Kit. A personal computer Asus, with 11th Gen Intel(R) Core(TM) i7-1165G7 and with 32 GB of RAM, was used as a high-resource fog node. It runs Windows 11 Pro OS as its operating system with JDK 1.8.0_333 to keep exported "jar" files compatible with JDK in Pi 4. A NodeMCU [58] development board was used as an end-device. It uses an ESP8266 [59] microcontroller with integrated capabilities for GPIO, PWM, IIC, 1-Wire, and ADC. Please see Figure 11 for hardware.

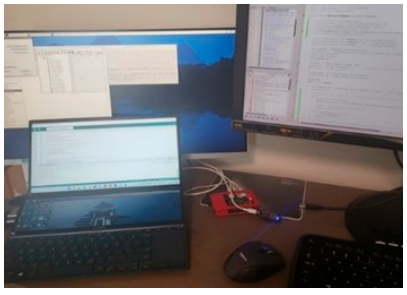

Figure 11. Experimental hardware set up.

JADE [60] was used as an agent development framework. It is an open-source platform for multi-agent Java applications. This framework offers a simple and powerful task execution, peer-to-peer agent communication using asynchronous messages, service discovery based on the yellow pages approach, and a possibility to integrate some add-ons and services. As a security solution, the JADE-S add-on was used. It enables user authentication, message signing, and encryption as an option.

Development environments that were used included Apache NetBeans for Java applications. Arduino IDE v2.0.3 was used for NodeMCU applications as an end-device using C++ programming language. Compiled Java applications were uploaded to Pi 4 using

WinSCP client application. Calculations were performed and charts were concluded mainly in Microsoft Excel. Occasionally, it was completed using Python in Visual Studio Code.

In order to measure the voltage and current, a USB tester UNI-T UT658B (Uni-Trend Technology (China) Co., Ltd., Dongguan City, China) was connected to a power adapter. As a second option, the energy meter PeakTech 9035 (PeakTech Prüf- und Messtechnik GmbH, Ahrensburg, Germany) was used for power calculations.

4.2. Experimental Metrics

In order to determine a response time in the experimental set up to identify how fast a solution is given by the orchestrator, a special class, which is called GlobalTimer, was developed. This class uses the method System.nanoTime(). It returns in nanoseconds the current value of a running Java Virtual Machine's high-resolution time source. This method is designed to measure only the elapsed time. To avoid any negative impact of logging events, logged events are stored only in RAM and printed as required only after the process is completed. Response time was used to measure the impact on performance made by a stress package overloading or security package as performance metrics. Please see Figure 12 for more details.

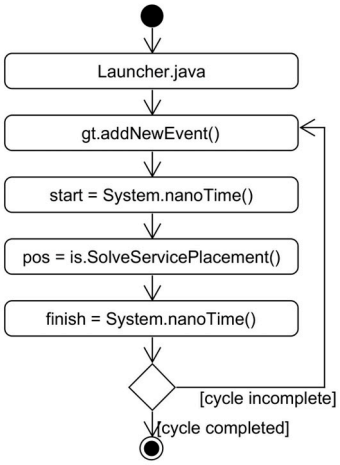

Figure 12. GlobalTimer class.

In order for PSO iterations to be successful, this has to lead to a convergence. As the publication [61] claims, convergence can be faster at the expense of a higher number recorded for errors. And it can have a lower number of errors, but the performance will be slower. Therefore, such a success rate can be used to evaluate the convergence performance of the metrics in an attempt to find the right balance.

While performing stress tests, a specifically designed stress test package was installed on Raspberry Pi 4. It allows us to gradually overload a CPU or RAM with a certain number of workers. Workers act as a workload, which increases with an increasing number of number of workers leaving only a certain percentage of resources unoccupied.

Specific calculations such as battery level sensing, placement finding, or service redistribution did not seem to have a diverse impact on power demands. The main factor was the duration of a specific process. Therefore, it was reasonable to use an electrical energy measurement (mW/h).

5. Results

A few types of experiments were performed to test the performance of the method that was presented in the publications [54,55]. Choosing the right coefficients such as an inertia weight means an optimal balance between the lowest number of global optimum failures and the shortest possible execution duration. A criteria number in its turn defines the range of a matrix for a pairwise comparison. A few criteria may initially be enough,

but the number may increase as the system complexity increases. Stress tests are meant to take into account the availability of resources, which may vary due to some background processes. A security package and its services trigger additional calculations. Therefore, it may or may not have a significant impact on the response time. And finally, due to the mobile nature of fog devices and their limited energy resources, power consumption is considered and tested.

For the IMOPSO algorithm to function as optimally as it is possible, it is necessary to consider such coefficients as inertia, cognitive, and social. Inertia weight was first introduced in [62]. The purpose of the inertia weight w is to balance between the local search and the global search. Please see Figure 13 for more details.

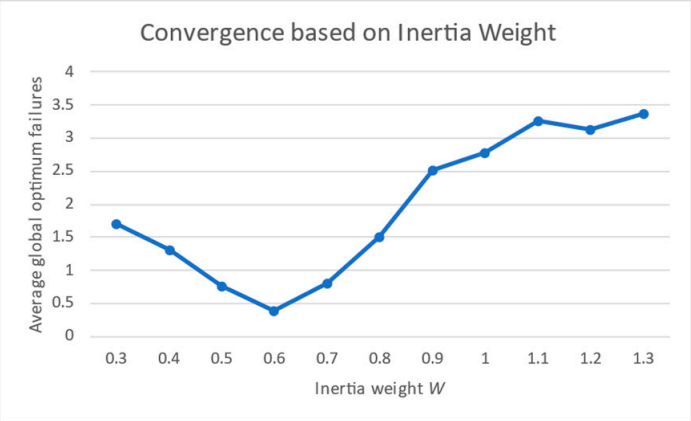

Figure 13. Inertia weight impact on convergence.

As a default setting, 4 simulated fog nodes, 12 services, and 200 epochs were used. Particles ranged from 50 to 500. The range of the inertia weight was 0.3 to 1.3. The cognitive and social coefficient was 1.499, which had a slight adjustment to the value of 1.496180. As per publication [63], it leads to convergent behavior. As is visible from the test results, the inertia weight coefficient 0.6 demonstrated the best outcome in finding a global optimum. It can be considered as a threshold value, which will be used in further experiments.

Different fog nodes might have different technical characteristics since heterogeneity is a part of fog computing. They also might have constrained resources and different QoS requirements. We use a judgement AHP matrix to evaluate such a situation and to prioritize different criteria. All the experiments are mainly completed using a four-criteria matrix. It includes power, CPU, security, and RAM. Power primarily, in our experiments, is expressively prioritized over other criteria. There is no particular reason for this, and any criteria can be adjusted as required.

$$Q = \begin{bmatrix} 1 & 7 & 7 & 7 \\ 1/7 & 1 & 2 & 2 \\ 1/7 & 1/2 & 1 & 2 \\ 1/7 & 1/2 & 1/2 & 1 \end{bmatrix} \quad (2)$$

The purpose of the experiment below is to test how much delay can a certain number of criteria contribute to our optimization algorithm. As a default setting, 4 simulated fog nodes, 12 services, 50 particles, and 200 epochs were used. A PC and a Raspberry Pi 4 are used as physical fog nodes to compare a powerful fog node running a simulation test and a fog node with limited hardware resources running a simulation test. Please see Figure 14 for more details.

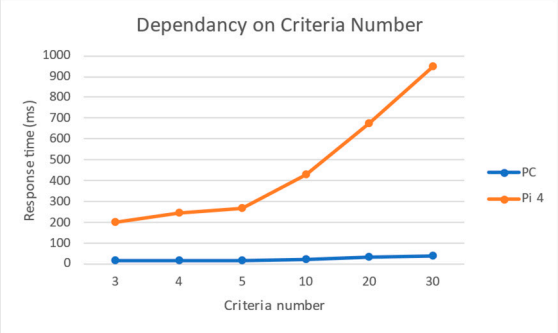

Figure 14. Algorithm optimization response time based on criteria number.

A range of 3 to 5 criteria is mostly expected. However, to test a broader scope, the range of 10 to 30 is included. Furthermore, 3 to 5 criteria generate only 3 to 10 comparisons within the matrix. This has little effect on the response time. However, 10 to 30 criteria generate 40 to 435 comparisons, and the response time significantly rises in a hardware-restricted fog node device. Still, a 1 s response time is manageable in real applications even though 30 criteria are barely needed.

The following experiment is meant to test the effect of CPU availability on the response time. The goal is to consider CPU utilization in such a case when additional services are to be hosted or some background processes take place, which overload the CPU. In order to learn the extent and a threshold to which a CPU can be overloaded by additional services, a CPU stress test was performed. Please see Figure 15 for more details.

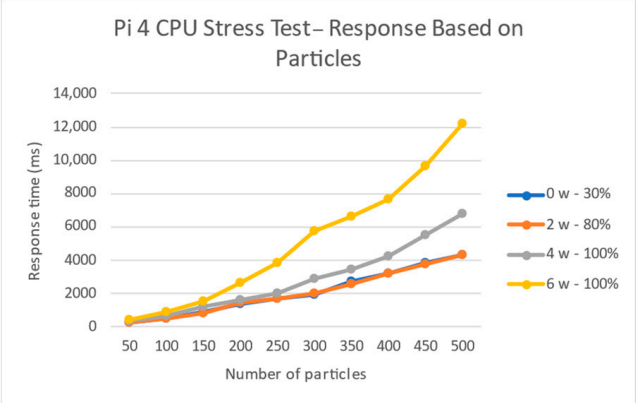

Figure 15. Algorithm optimization response time based on CPU overloading.

Raspberry Pi 4 was used as a physical fog node. The optimization algorithm performance evaluation was completed using 4 simulated fog nodes, 12 services, and 200 epochs as a default setting. The number of particles ranges from 50 to 500. To perform a CPU stress test, a stress tool was installed. A CPU was stressed with 0 to 6 workers, which means that concurrent process threads that are launched in CPU overload it. The algorithm itself overloads Pi 4 CPU at around 30% once it is started. In total, 0 to 2 workers do not seem to pose any negative effect on a CPU. However, 4 to 6 workers that overload a CPU up to 100% are critical. Services would have to be redistributed before such a level is reached.

The following experiment is meant to test the effect of RAM availability. The optimization algorithm performance evaluation was completed using 4 simulated fog nodes, 12 services, and 200 epochs as a default setting. The number of particles ranges from 50 to 500. Please see Figure 16 for more details.

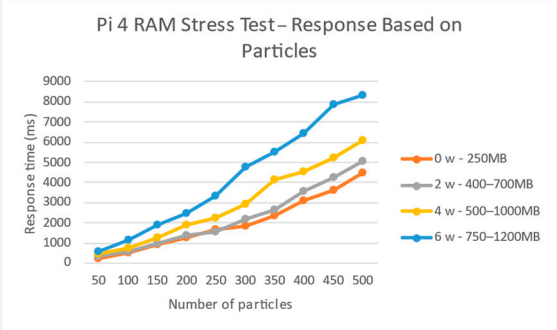

Figure 16. Algorithm optimization response time based on RAM overloading.

Raspberry Pi 4 background processes initially overload RAM with about 220 MB. Once the algorithm is launched, the level goes up to 250 MB. It suggests that the algorithm is not that dependent on RAM, as it is dependent on CPU. However, if there are some significant background processes or additional services, it can still slow down the algorithm and the fog node services need a redistribution.

The following experiment is completed to test a low battery level. An external analog-to-digital module would be needed for this since Raspberry Pi 4 does not natively offer an ADC capability to detect a certain level of voltage. However, a low or high input was used for the test purposes. The class BattVolt.java was created to test a battery voltage level. The library P4J was used to read an input value by the BattVolt agent. The polling interval by default was 5000 ms.

Using Raspberry Pi 4 as a low-resource fog node, it takes up to 1.5 s to complete the whole process of a service redistribution. Using a PC as a powerful fog node, it would require about 10 times lower duration. More details are available in the paper [54]. CPU overloading does not seem to affect battery level sensing or service redistribution. However, a two-stage IMOPSO and AHP algorithm is sensitive enough to overloading. Still, a few seconds to redistribute fog node services due to a low battery level might not be a problem in real-life conditions. Please see Figure 17 for graphical details and Table 2 for measurement details.

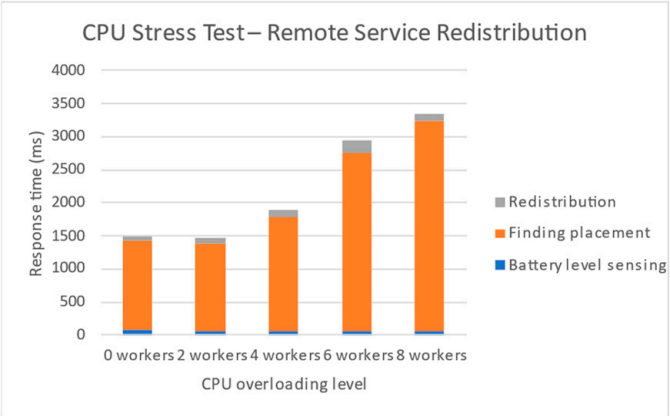

Figure 17. Service redistribution due to low battery charge.

Table 2. Service redistribution response time (ms).

Process	0 Workers	2 Workers	4 Workers	6 Workers	8 Workers
Battery level sensing	75	60	68	52	67
Finding placement	1359	1326	1723	2687	3167
Redistribution	63	78	97	202	113
Total time	1497	1464	1888	2951	3347

The following five experiments are meant to test the security influence on a fog node response time using different security levels and two fog nodes with different hardware capabilities. JADE-S, a security add-on package, was used as a security option [64]. This add-on allows us to develop multi-agent applications with a certain degree of security, including guaranteed message integrity, confidentiality, and authorization checks. It allows for agents and containers in a platform to be owned by authenticated users, which are authorized by a platform administrator. Each agent has a public and a private key pair, which is used to sign and encrypt messages.

The JADE security guide claims that the signing and encrypting of messages can slow down the agent communication performance and that this is the reason why it is not completed by default; it is important to check to what extent it can happen. An optimization algorithm performance evaluation was completed using 2 physical fog nodes and 12 services as a default setting. Services have to be moved from one fog node to another due to a low battery. The SecurityHeper() package is used with the methods setUseSignature() and setUseEncryption() to set a signature and encryption for a message. The methods getUseSignature() and getUseEncryption() of the same package are used to retrieve a signature and encryption. Default configuration values are used such as an RSA asymmetric algorithm and a 512-bit key size for public and private keys.

As the chart above in Figure 18 suggests, four experiments with different security configurations were performed. It includes (a) no security package, (b) signed, (c) encrypted, and (d) signed and encrypted security levels. A PC was used as a high-resource fog node and services had to be moved from one fog node to another due to a low battery level. Such a remote service redistribution did not demonstrate any significant response time variation because of different security approaches. There is a slight response time increase mainly due to a battery level sensing time increase.

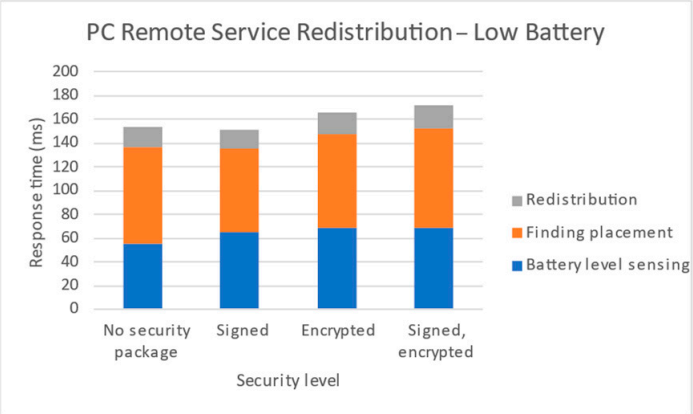

Figure 18. Security package impact on communication in high-resource nodes.

When a low-resource fog node is used such as Raspberry Pi 4, the response time increases by almost 10 times, as is observed in Figure 19 above. It increases mainly by an impact of the placement finding algorithm, which is mostly resource prone as was witnessed in the publication [54]. The usage of a security package does not seem to have any significant impact, apparently because of no communication between agents. The response time may increase slightly in redistribution and battery level sensing during their turn, as these processes involve a sensed state communication to a Decision Maker and the communication of a decision made by a Decision Maker to Execution agents and finally Synchronization agents.

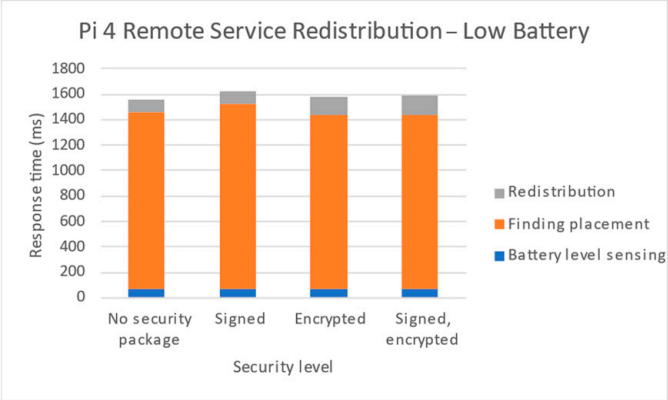

Figure 19. Security package impact on communication in low-resource nodes.

In order to increase a security level, security parameters can be adjusted instead of using default ones. This can be achieved by launching addition services. The following experiment is meant to test the impact of a key size on the response time. The same scenario of remote service redistribution due to a low battery is used. Raspberry Pi 4 and a PC were used as two different fog nodes. Three different key length configurations were used with each fog node. The service jade.security.AsymKeySize has to be launched and the key length options are 512, 1024, and 2048.

As it is visible in Figure 20, the size for public and private keys did not have any significant impact on agent communication performance. There might be some slight increase in time due to procedures that involve more communication of agents, as in redistribution and battery level sensing, but the biggest impact is made by the placement finding method.

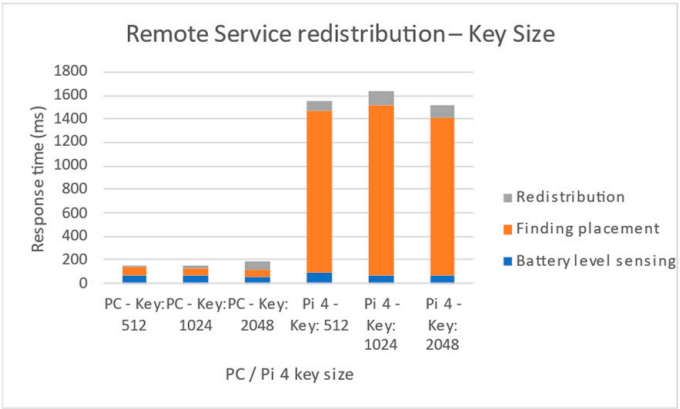

Figure 20. Response time dependency on an asymmetric key size.

Messages can be intercepted and read if they are exchanged as plain text. Therefore, additional measures to encrypt them are beneficial. The following experiment is meant to test the impact of a message encryption algorithm on the response time. The same scenario of remote service redistribution due to a low battery is used with Raspberry Pi 4 and a PC as two different fog nodes. Four different symmetric algorithm configurations were used for messages with each fog node. The service jade.security.SymAlgorithm has to be launched and the chosen options were AES, Blowfish, DES, and TripleDES. More details are available in Figure 21.

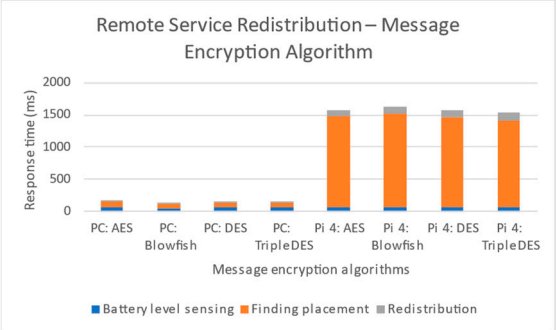

Figure 21. Response time dependency on message encryption.

As is visible from the chart above, a message encryption algorithm did not have any significant impact. One of potential reasons might be the messages themselves that are being communicated, since they are very short. Short messages might not be resource demanding. Long messages would give a better idea, but these are simple agent communication commands and no increase in their length is needed.

The following experiment is meant to test the impact of a key pair algorithm on the response time. Two different asymmetric algorithm configurations were used to generate key pairs with each fog node. The service jade.security.AsymAlgorithm has to be launched and the options to choose from are RSA, which is the default one, and DSA. More details are available in Figure 22 below.

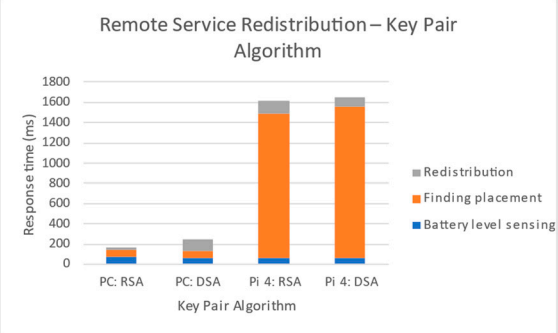

Figure 22. Response time dependency on an asymmetric key pair algorithm.

RSA and DSA do not seem to have any significant advantage over each other, allowing us to conclude that the usage of security add-on does not contribute to a higher response time, as is suggested in the Jade security guide. Moreover, the usage of other security configurations rather than default ones does not obviously contribute to a higher response time either.

It is important to maximize the overall runtime of the system. Mobile fog nodes are known for scarce power supply and therefore resources have to be used in an optimal way. The best service distribution from the perspective of power is the one used when nodes are evenly loaded to keep the whole system available as long as possible.

The following experiment is meant to measure the power in watts that is needed to obtain a service placement solution. For the method to be physically available, it has to be running in a fog node. Raspberry Pi 4 was used as a host for the JADE platform. Measurements were made solely with a running Pi 4, and with launched JADE working on its tasks in the Pi 4. Power needs were compared with a regular 40 W bulb for illustrative purposes. An official power supply adapter with the output of 5.1 V and 3.0 A was used. The USB tester UNI-T UT658B was used to measure the voltage and amperage, which served as a power input for Pi 4. Please see Figure 23 for more details.

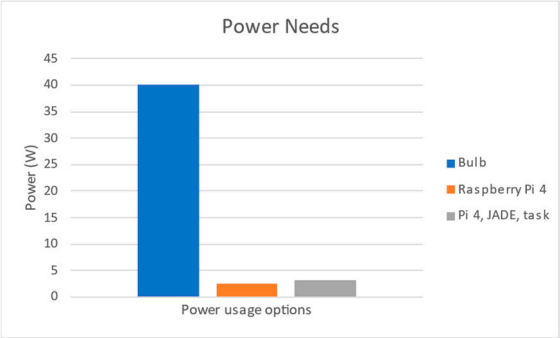

Figure 23. Service placement power consumption.

A remote service redistribution scenario was used. In total, 5.1 V and 0.5 A were needed to keep Raspberry Pi 4 running. It is equal to 2.55 W of power. If JADE was launched and no optimization algorithms were running, power consumption settled to the same 2.55 W after a few seconds. However, once an optimization process begins, voltage stays more or less the same but the current increases to 0.62 A, which is equal to 3.162 W. More details are given in Table 3 below.

Table 3. Energy consumption comparison.

Options	Energy (Wh)	Price (cnt/h)	Price (cnt/30 Days)
Bulb	40	0.856	616.32
Raspberry Pi 4	2.55	0.055	39.29
Pi 4, JADE, placement	3.162	0.068	48.72

A very tiny amount of energy is needed and the costs to maintain such a fog node are minimal as well. Even if the optimization process is running without any interruptions for the whole month, which is barely required in real-life conditions, self-cost will be around just 0.1 EUR, considering that the price for electricity is 0.214 EUR/kWh. However, power demands did not vary on the type of tasks that were running such as battery level sensing, service placement finding, or redistribution. This suggests that only the active time of the optimization should be considered as the whole instead of breaking it into different processes.

The following experiment was conducted to analyze a service placement solution energy consumption based on task scalability. An optimization model was run 20 times with each parameter set using Raspberry Pi 4 as a fog node. Fog computing system complexity was gradually increased by increasing the number of fog nodes, services, particles, and epochs. An average value was used of each 20 attempts. Please see Figure 24 and Table 4 for more details.

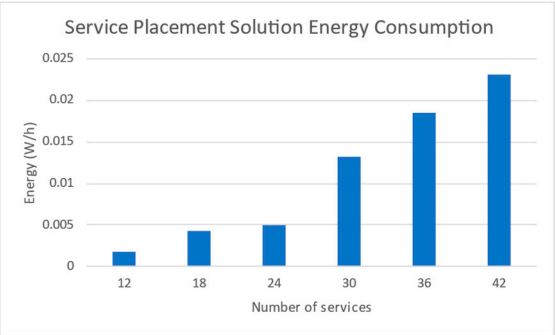

Figure 24. Scalability impact on the energy consumption.

Table 4. Scalability parameter sets.

Nodes	Services	Particles	Epochs	Response (s)	Energy (W/h)
4	12	200	300	2.032	0.00178
6	18	300	300	4.853	0.00426
8	24	400	400	5.72	0.00502
10	30	500	600	15.082	0.01325
12	36	600	700	21.118	0.01855
14	42	600	800	26.305	0.02310

As is visible from Table 4, finding a service placement for 14 nodes and 42 services requires a very small amount of consumed energy, which is around just 23 mW/h. This is barely considerable, but what requires more attention is the time itself. A response time of over 20 s is unacceptable in field conditions. However, the calculation response time can be improved by using a more powerful fog node such as PC. With the current complexity configuration and low-resource fog node, it can be still beneficial to use it with up to 6 nodes and 18 services with a response time of a few seconds. A bigger problem to address in this case would be keeping a mobile fog node running on a battery for a while when its power supply is limited.

6. Discussion

Different experiments were conducted with the prototype to determine the usability of the fog service placement orchestrator as a multi-agent system. The JADE platform was used to implement fog computing services as agents in Java language. A PC was used as a high-resource fog node, and Pi 4 stood for a low-resource node. As for an increased platform security, the JADE-S security add-on was integrated. A Raspberry Pi 4 stress package was installed for a stress test, and measurements were taken for energy consumption. Additionally, the inertia weight coefficient and a matrix size test were completed to determine the best performance choices.

The inertia weight, according to the publication [63], demonstrates a clear balanced relationship between an exploration and an exploitation. It is possible to avoid a premature convergence by choosing the right inertia weight w [65]. The inertia weight of 0.6 showed the best results, since it leads, on average, to the lowest number of failures in finding the global optimum. Meanwhile, the number of criteria is used for a judgement matrix to highlight the importance of certain qualities such as security, CPU, RAM, and power. Four criteria were used for all other experiments but, as is visible from the results of the current study, even 30 criteria with 435 comparisons have a barely visible impact on a high-resource fog node performance. As for a low-resource fog node and a 30-criteria matrix, a service placement can be found in less than 1 s, which is still usable in field conditions.

It is possible to conclude based on the CPU stress test that at least 20% of CPU must be available for calculations. Otherwise, the response time increases two or three times. It is not that relevant if the number of particles is low, suggesting that the calculations are relatively simple, but it becomes critical with more complex ones. A CPU stress test has the biggest impact on the service placement phase in contrast to other ones that are relatively simple. When it increases from 1.1 s to 3.2 s, it may have a considerable negative effect on tasks that require a short response time.

The security add-on JADE-S did not add any clearly observed delay. Service placement decision making takes most of the time, but it does not require any agent intercommunication that is signed, encrypted, or both. Other processes are less complicated and less time-consuming. Some spontaneous time increases or decreases were noted but they were irregular and did not seem to play a significant role.

A USB tester and an energy meter were used to measure the voltage and current. Power or energy were measured or calculated as required. A self-const of the JADE platform calculations for 1 month would make only around 0.1 EUR, and the availability of Raspberry Pi 4 as a hardware would add another 0.4 EUR. However, mobility requires

one to use batteries, and even such energy needs can be an issue when the capacity of the batteries is low enough.

The experimental results allow us to claim that this solution is applicable in a small fog-computing infrastructure. It offers dynamic and distributed decision making. A distributed decision-making process is more resilient than a central architecture due to the absence of a single point of failure. Any fog node can make decisions to launch, place, and eliminate relevant services. These decisions are synchronized afterwards. If there are any failures, services can be relocated. It is also possible to sign and to encrypt agent messages for additional security. Constant resource monitoring allows us to keep track of the available resources to make informed decisions.

7. Conclusions

A new optimization method was introduced for an optimized service placement. A two-stage method uses the IMOPSO algorithm to find a Pareto optimal set of potential service placements. Meanwhile, the AHP algorithm makes a final choice among the potential placements using a judgement matrix of priorities. It gives an assessment based on the Eigen vector in relation to the considered criteria.

The major novelty of this research is a new service placement orchestration method as a multi-agent system. Multi-agent systems are vulnerable to network attacks, and they may lack an integrated monitoring tool. Security issues were addressed by integrating a JADE-S add-on, which allows us to sign and encrypt agent communication messages. Monitoring classes were developed to track resources. In addition to this, there is no single central control unit to avoid a single point of failure. Decisions are made in a distributed and dynamic approach. Decisions can be made by a fog node that is close enough to a user and the outcome is therefore synchronized with other fog nodes. Distributed decision making contributes to resilience.

Multiple tests were performed to determine prototype usability including dependency on the inertia weight and the number of criteria, stress tests, security package performance impact, and power consumption. These tests witness the usability of the method in field conditions with a security package with no additional obvious delay. Power needs are low and the maintenance of the system is cheap. The main limitation is a response time with low-resource fog nodes, but it is still applicable for a few fog node infrastructures. The infrastructure with high-resource fog nodes can be a few times larger.

Author Contributions: Conceptualization, N.Š. and A.V.; methodology, N.Š. and A.V.; software, N.Š.; validation, N.Š.; formal analysis, N.Š.; investigation, N.Š.; resources, A.V.; data curation, N.Š.; writing—original draft preparation, N.Š.; writing—review and editing, N.Š.; visualization, N.Š.; supervision, A.V.; project administration, A.V.; funding acquisition, A.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Dataset available on request from the authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Srirama, S.N. A Decade of Research in Fog Computing: Relevance, Challenges, and Future Directions. *Softw. Pract. Exp.* **2024**, *54*, 3–23. [CrossRef]
2. Mu, X.; Antwi-Afari, M.F. The Applications of Internet of Things (IoT) in Industrial Management: A Science Mapping Review. *Int. J. Prod. Res.* **2024**, *62*, 1928–1952. [CrossRef]
3. Hazra, A.; Rana, P.; Adhikari, M.; Amgoth, T. Fog Computing for Next-Generation Internet of Things: Fundamental, State-of-the-Art and Research Challenges. *Comput. Sci. Rev.* **2023**, *48*, 100549. [CrossRef]
4. Das, R.; Inuwa, M.M. A Review on Fog Computing: Issues, Characteristics, Challenges, and Potential Applications. *Telemat. Inform. Rep.* **2023**, *10*, 100049. [CrossRef]
5. Mirampalli, S.; Wankar, R.; Srirama, S.N. Evaluating NiFi and MQTT Based Serverless Data Pipelines in Fog Computing Environments. *Future Gener. Comput. Syst.* **2024**, *150*, 341–353. [CrossRef]

6. Aldossary, M. Multi-Layer Fog-Cloud Architecture for Optimizing the Placement of IoT Applications in Smart Cities. *Comput. Mater. Contin.* **2023**, *75*, 633–649. [CrossRef]
7. Aqib, M.; Kumar, D.; Tripathi, S. Machine Learning for Fog Computing: Review, Opportunities and a Fog Application Classifier and Scheduler. *Wirel. Pers Commun* **2023**, *129*, 853–880. [CrossRef]
8. Tran-Dang, H.; Kim, D.-S. Fog Computing: Fundamental Concepts and Recent Advances in Architectures and Technologies. In *Cooperative and Distributed Intelligent Computation in Fog Computing: Concepts, Architectures, and Frameworks*; Tran-Dang, H., Kim, D.-S., Eds.; Springer Nature: Cham, Switzerland, 2023; pp. 1–18. ISBN 978-3-031-33920-2.
9. Dastjerdi, A.V.; Gupta, H.; Calheiros, R.N.; Ghosh, S.K.; Buyya, R. Chapter 4—Fog Computing: Principles, Architectures, and Applications. In *Internet of Things*; Buyya, R., Vahid Dastjerdi, A., Eds.; Morgan Kaufmann: Burlington, MA, USA, 2016; pp. 61–75. ISBN 978-0-12-805395-9.
10. Waqas, M.; Tu, S.; Wan, J.; Mir, T.; Alasmary, H.; Abbas, G. Defense Scheme against Advanced Persistent Threats in Mobile Fog Computing Security. *Comput. Netw.* **2023**, *221*, 109519. [CrossRef]
11. Aggarwal, S.; Kumar, N. Fog Computing for 5G-Enabled Tactile Internet: Research Issues, Challenges, and Future Research Directions. *Mob. Netw. Appl.* **2023**, *28*, 690–717. [CrossRef]
12. Vu Khanh, Q.; Vi Hoai, N.; Dang Van, A.; Nguyen Minh, Q. An Integrating Computing Framework Based on Edge-Fog-Cloud for Internet of Healthcare Things Applications. *Internet Things* **2023**, *23*, 100907. [CrossRef]
13. Burhan, M.; Alam, H.; Arsalan, A.; Rehman, R.A.; Anwar, M.; Faheem, M.; Ashraf, M.W. A Comprehensive Survey on the Cooperation of Fog Computing Paradigm-Based IoT Applications: Layered Architecture, Real-Time Security Issues, and Solutions. *IEEE Access* **2023**, *11*, 73303–73329. [CrossRef]
14. Dorri, A.; Kanhere, S.S.; Jurdak, R. Multi-Agent Systems: A Survey. *IEEE Access* **2018**, *6*, 28573–28593. [CrossRef]
15. Wang, J.; Deng, X.; Guo, J.; Zeng, Z. Resilient Consensus Control for Multi-Agent Systems: A Comparative Survey. *Sensors* **2023**, *23*, 2904. [CrossRef] [PubMed]
16. Jayanagara, O.; Wuisan, D.S.S. An Overview of Concepts, Applications, Difficulties, Unresolved Issues in Fog Computing and Machine Learning. *Int. Trans. Artif. Intell.* **2023**, *1*, 213–229. [CrossRef]
17. Apat, H.K.; Nayak, R.; Sahoo, B. A Comprehensive Review on Internet of Things Application Placement in Fog Computing Environment. *Internet Things* **2023**, *23*, 100866. [CrossRef]
18. Lahmar, I.B.; Boukadi, K. Resource Allocation in Fog Computing: A Systematic Mapping Study. In Proceedings of the 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), Paris, France, 20–23 April 2020; pp. 86–93.
19. Saif, F.A.; Latip, R.; Hanapi, Z.M.; Shafinah, K. Multi-Objective Grey Wolf Optimizer Algorithm for Task Scheduling in Cloud-Fog Computing. *IEEE Access* **2023**, *11*, 20635–20646. [CrossRef]
20. Ogundoyin, S.O.; Kamil, I.A. Optimal Fog Node Selection Based on Hybrid Particle Swarm Optimization and Firefly Algorithm in Dynamic Fog Computing Services. *Eng. Appl. Artif. Intell.* **2023**, *121*, 105998. [CrossRef]
21. Saif, F.A.; Latip, R.; Hanapi, Z.M.; Alrshah, M.A.; Kamarudin, S. Workload Allocation toward Energy Consumption-Delay Trade-Off in Cloud-Fog Computing Using Multi-Objective NPSO Algorithm. *IEEE Access* **2023**, *11*, 45393–45404. [CrossRef]
22. Ibrahim, M.A.; Askar, S. An Intelligent Scheduling Strategy in Fog Computing System Based on Multi-Objective Deep Reinforcement Learning Algorithm. *IEEE Access* **2023**, *11*, 133607–133622. [CrossRef]
23. Tran-Dang, H.; Kim, D.-S. DISCO: Distributed Computation Offloading Framework for Fog Computing Networks. *J. Commun. Netw.* **2023**, *25*, 121–131. [CrossRef]
24. Islam, M.M.; Ramezani, F.; Lu, H.Y.; Naderpour, M. Optimal Placement of Applications in the Fog Environment: A Systematic Literature Review. *J. Parallel Distrib. Comput.* **2023**, *174*, 46–69. [CrossRef]
25. Righi, R. *Scheduling Problems: New Applications and Trends*; BoD–Books on Demand: London, UK, 2020; ISBN 978-1-78985-053-6.
26. Pallewatta, S.; Kostakos, V.; Buyya, R. Placement of Microservices-Based IoT Applications in Fog Computing: A Taxonomy and Future Directions. *ACM Comput. Surv.* **2023**, *55*, 321:1–321:43. [CrossRef]
27. Zare, M.; Elmi Sola, Y.; Hasanpour, H. Towards Distributed and Autonomous IoT Service Placement in Fog Computing Using Asynchronous Advantage Actor-Critic Algorithm. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 368–381. [CrossRef]
28. Salaht, F.A.; Desprez, F.; Lebre, A. An Overview of Service Placement Problem in Fog and Edge Computing. *ACM Comput. Surv.* **2021**, *53*, 1–35. [CrossRef]
29. Ostrowski, K.; Małecki, K.; Dziurzański, P.; Singh, A.K. Mobility-Aware Fog Computing in Dynamic Networks with Mobile Nodes: A Survey. *J. Netw. Comput. Appl.* **2023**, *219*, 103724. [CrossRef]
30. Mahmud, R.; Ramamohanarao, K.; Buyya, R. Latency-Aware Application Module Management for Fog Computing Environments. *ACM Trans. Internet Technol.* **2018**, *19*, 9:1–9:21. [CrossRef]
31. Chen, M.; Li, W.; Fortino, G.; Hao, Y.; Hu, L.; Humar, I. A Dynamic Service Migration Mechanism in Edge Cognitive Computing. *ACM Trans. Internet Technol.* **2019**, *19*, 30:1–30:15. [CrossRef]
32. Filiposka, S.; Mishev, A.; Gilly, K. Mobile-Aware Dynamic Resource Management for Edge Computing. *Trans. Emerg. Telecommun. Technol.* **2019**, *30*, e3626. [CrossRef]
33. Mseddi, A.; Jaafar, W.; Elbiaze, H.; Ajib, W. Joint Container Placement and Task Provisioning in Dynamic Fog Computing. *IEEE Internet Things J.* **2019**, *6*, 10028–10040. [CrossRef]
34. Jošilo, S.; Dán, G. Decentralized Algorithm for Randomized Task Allocation in Fog Computing Systems. *IEEE/ACM Trans. Netw.* **2019**, *27*, 85–97. [CrossRef]

35. Zhu, C.; Tao, J.; Pastor, G.; Xiao, Y.; Ji, Y.; Zhou, Q.; Li, Y.; Ylä-Jääski, A. Folo: Latency and Quality Optimized Task Allocation in Vehicular Fog Computing. *IEEE Internet Things J.* **2019**, *6*, 4150–4161. [CrossRef]
36. Aral, A.; Ovatman, T. A Decentralized Replica Placement Algorithm for Edge Computing. *IEEE Trans. Netw. Serv. Manag.* **2018**, *15*, 516–529. [CrossRef]
37. Lee, G.; Saad, W.; Bennis, M. An Online Optimization Framework for Distributed Fog Network Formation with Minimal Latency. *IEEE Trans. Wirel. Commun.* **2019**, *18*, 2244–2258. [CrossRef]
38. Zhao, H.; Wang, S.; Shi, H. Fog-Computing Based Mobility and Resource Management for Resilient Mobile Networks. *High-Confid. Comput.* **2023**, *4*, 100193. [CrossRef]
39. Ebrahim, M.; Hafid, A. Resilience and Load Balancing in Fog Networks: A Multi-Criteria Decision Analysis Approach. *Microprocess. Microsyst.* **2023**, *101*, 104893. [CrossRef]
40. Pallewatta, S.; Kostakos, V.; Buyya, R. MicroFog: A Framework for Scalable Placement of Microservices-Based IoT Applications in Federated Fog Environments. *J. Syst. Softw.* **2024**, *209*, 111910. [CrossRef]
41. Núñez-Gómez, C.; Carrión, C.; Caminero, B.; Delicado, F.M. S-HIDRA: A Blockchain and SDN Domain-Based Architecture to Orchestrate Fog Computing Environments. *Comput. Netw.* **2023**, *221*, 109512. [CrossRef]
42. Dogani, J.; Yazdanpanah, A.; Zare, A.; Khunjush, F. A Two-Tier Multi-Objective Service Placement in Container-Based Fog-Cloud Computing Platforms. *Clust. Comput* **2023**, 1–24. [CrossRef]
43. Sofia, R.C.; Dykeman, D.; Urbanetz, P.; Galal, A.; Dave, D.A. Dynamic, Context-Aware Cross-Layer Orchestration of Containerized Applications. *IEEE Access* **2023**, *11*, 93129–93150. [CrossRef]
44. Cheng, J.; Nguyen, D.T.; Bhargava, V.K. Resilient Edge Service Placement under Demand and Node Failure Uncertainties. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 558–573. [CrossRef]
45. Azizi, S.; Shojafar, M.; Farzin, P.; Dogani, J. DCSP: A Delay and Cost-Aware Service Placement and Load Distribution Algorithm for IoT-Based Fog Networks. *Comput. Commun.* **2024**, *215*, 9–20. [CrossRef]
46. Singh, S.; Vidyarthi, D.P. An Integrated Approach of ML-Metaheuristics for Secure Service Placement in Fog-Cloud Ecosystem. *Internet Things* **2023**, *22*, 100817. [CrossRef]
47. Chouat, H.; Abbassi, I.; Graiet, M.; Südholt, M. Adaptive Configuration of IoT Applications in the Fog Infrastructure. *Computing* **2023**, *105*, 2747–2772. [CrossRef]
48. Zare, M.; Sola, Y.E.; Hasanpour, H. Imperialist Competitive Based Approach for Efficient Deployment of IoT Services in Fog Computing. *Clust. Comput* **2023**, *27*, 845–858. [CrossRef]
49. Amjad, S.; Akhtar, A.; Ali, M.; Afzal, A.; Shafiq, B.; Vaidya, J.; Shamail, S.; Rana, O. Orchestration and Management of Adaptive IoT-Centric Distributed Applications. *IEEE Internet Things J.* **2024**, *11*, 3779–3791. [CrossRef] [PubMed]
50. Isa, I.S.M.; El-Gorashi, T.E.H.; Musa, M.O.I.; Elmirghani, J.M.H. Resilient Energy Efficient IoT Infrastructure with Server and Network Protection for Healthcare Monitoring Applications. *IEEE Access* **2024**, *12*, 48910–48940. [CrossRef]
51. Mutlag, A.A.; Ghani, M.K.A.; Mohammed, M.A.; Lakhan, A.; Mohd, O.; Abdulkareem, K.H.; Garcia-Zapirain, B. Multi-Agent Systems in Fog–Cloud Computing for Critical Healthcare Task Management Model (CHTM) Used for ECG Monitoring. *Sensors* **2021**, *21*, 6923. [CrossRef] [PubMed]
52. Jain, M.; Saihjpal, V.; Singh, N.; Singh, S.B. An Overview of Variants and Advancements of PSO Algorithm. *Appl. Sci.* **2022**, *12*, 8392. [CrossRef]
53. Costa, D.S.; Mamede, H.S.; da Silva, M.M. A Method for Selecting Processes for Automation with AHP and TOPSIS. *Heliyon* **2023**, *9*, e13683. [CrossRef] [PubMed]
54. Liutkevičius, A.; Morkevičius, N.; Venčkauskas, A.; Toldinas, J. Distributed Agent-Based Orchestrator Model for Fog Computing. *Sensors* **2022**, *22*, 5894. [CrossRef] [PubMed]
55. Morkevicius, N.; Venčkauskas, A.; Šatkauskas, N.; Toldinas, J. Method for Dynamic Service Orchestration in Fog Computing. *Electronics* **2021**, *10*, 1796. [CrossRef]
56. Coello, C.A.C.; Pulido, G.T.; Lechuga, M.S. Handling Multiple Objectives with Particle Swarm Optimization. *IEEE Trans. Evol. Comput.* **2004**, *8*, 256–279. [CrossRef]
57. Saaty, R.W. The Analytic Hierarchy Process—What It Is and How It Is Used. *Math. Model.* **1987**, *9*, 161–176. [CrossRef]
58. NodeMcu—An Open-Source Firmware Based on ESP8266 Wifi-Soc. Available online: https://www.nodemcu.com/index_en.html#fr_54745c8bd775ef4b99000011 (accessed on 14 May 2024).
59. ESP8266 Wi-Fi SoC | Espressif Systems. Available online: https://www.espressif.com/en/products/socs/esp8266 (accessed on 14 May 2024).
60. Jade Site | Java Agent DEvelopment Framework. Available online: https://jade.tilab.com/ (accessed on 2 August 2022).
61. Modiri, A.; Kiasaleh, K. Modification of Real-Number and Binary PSO Algorithms for Accelerated Convergence. *IEEE Trans. Antennas Propagat.* **2011**, *59*, 214–224. [CrossRef]
62. Shi, Y.; Eberhart, R. A Modified Particle Swarm Optimizer. In Proceedings of the 1998 IEEE International Conference on Evolutionary Computation Proceedings, IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360), Anchorage, AK, USA, 4–9 May 1998; pp. 69–73.
63. Harrison, K.R.; Engelbrecht, A.P.; Ombuki-Berman, B.M. Inertia Weight Control Strategies for Particle Swarm Optimization. *Swarm Intell.* **2016**, *10*, 267–305. [CrossRef]

64. Tutorials & Guides | Jade Site. Available online: https://jade.tilab.com/documentation/tutorials-guides/ (accessed on 15 April 2024).
65. Wang, J.; Wang, X.; Li, X.; Yi, J. A Hybrid Particle Swarm Optimization Algorithm with Dynamic Adjustment of Inertia Weight Based on a New Feature Selection Method to Optimize SVM Parameters. *Entropy* **2023**, *25*, 531. [CrossRef] [PubMed]