electronics

MDPI

Article

Enabling Processing Power Scalability with Internet of Things (IoT) Clusters

Jorge Coelho [1,2,*] and Luís Nogueira [1]

[1] School of Engineering (ISEP), Polytechnic of Porto (IPP), 4249-015 Porto, Portugal; lmn@isep.ipp.pt
[2] Artificial Intelligence and Computer Science Laboratory, University of Porto (LIACC), 4099-002 Porto, Portugal
* Correspondence: jmn@isep.ipp.pt; Tel.: +351-228-340-500

Abstract: Internet of things (IoT) devices play a crucial role in the design of state-of-the-art infrastructures, with an increasing demand to support more complex services and applications. However, IoT devices are known for having limited computational capacities. Traditional approaches used to offload applications to the cloud to ease the burden on end-user devices, at the expense of a greater latency and increased network traffic. Our goal is to optimize the use of IoT devices, particularly those being underutilized. In this paper, we propose a pragmatic solution, built upon the Erlang programming language, that allows a group of IoT devices to collectively execute services, using their spare resources with minimal interference, and achieving a level of performance that otherwise would not be met by individual execution.

Keywords: edge computing; computational offloading; orchestration; IoT; functional programming

Citation: Coelho, J.; Nogueira, L. Enabling Processing Power Scalability with Internet of Things (IoT) Clusters. *Electronics* **2022**, *11*, 81. https://doi.org/10.3390/electronics11010081

Academic Editor: Akash Kumar

Received: 22 November 2021
Accepted: 24 December 2021
Published: 28 December 2021

1. Introduction

Given the ubiquity of internet of things (IoT) devices and their strong proliferation [1] many opportunities appear in exploring their potentialities, namely their connectivity and computational power [2]. It is well known that the quantity of data produced by a variety of data sources and sent to end systems to further processing is growing significantly, increasingly demanding more processing power. The challenges become even more critical when a coordinated content analysis of the data sent from multiple sources is necessary. Thus, with a potentially unbounded amount of stream data and limited resources, some of the processing tasks may not be satisfyingly answered by individual devices, guaranteeing a desired level of performance.

Computation offloading is recognized as a promising solution by migrating a part or an entire application to a remote server in order to be executed there. Various models and frameworks have been proposed to offload resource-intensive components of applications for more efficient execution [3–5]. However, these solutions rely on the concept of offloading to the cloud. Due to the increasing hardware capabilities of IoT devices and their proliferation, making it common to have several of these devices in the same area, offloading to the cloud may not always be a necessity, if the available resources of these devices are wisely used. The study of scenarios where heterogeneous nodes with unknown resources are aggregated in a collaborative effort to achieve some goal has been the subject of works such as [6] where some sort of data analysis is needed to estimate each node capacity and distribute work wisely.

Orchestration in distributed systems is a common approach to creating an abstraction layer between the several devices of the system. With the orchestration layer, devices that constitute the distributed system are "hidden" and their details and behavior are managed by the orchestrator (also referred as coordinator), providing a simplified interface to those devices and centered in the use of their resources without the need to know other

operational details. This is particularly relevant in service-oriented architectures (SOA) [7] with obvious applications when using clusters of IoT devices [8].

At the same time, functional programming is an established approach to implement parallel and distributed systems [9]. The minimization of the need of a shared state enables code distribution and parallel processing that fosters the development of easily scalable systems. Due to the rise of multicore and distributed systems, functional programming spread its influence through many mainstream languages [10,11] and is used in major cloud infrastructures such as the AWS Lambda [12]. In particular, Erlang [13], due to its simplicity and strong support for fault-tolerant distributed programming, is seen as a promising language for IoT applications [14].

In this paper, we propose an Erlang-based framework for the parallel processing of tasks in a cluster of IoT devices. These devices are able to communicate and report their resource availability, accepting computational tasks for execution. The goal is to have one of the connected devices requesting the offloading of tasks and relying on a module that coordinates all the communication process and balances the scheduling of tasks based on their estimated computational cost and the device's computational power and availability.

Resource allocation is one of the most complex problems in large multi-processor and distributed systems, and in general it is considered NP-hard. Computation platforms now integrate hundreds to thousands of processing cores, running complex and dynamic applications that make it difficult to foresee the amount of load they can impose to those platforms. Elementary combinatorics provides us with evidence of the problem of scale. For a simple formulation of the problem of allocating jobs to processors (one-to-one allocation), one can see that the number of allocations grows with the factorial of the number of jobs and processors.

A static allocation decided before deployment, based on the (nearly) complete knowledge about the load and the platform, is no longer viable. In traditional embedded systems, the workload is usually allocated in terms of its worst-case behaviour, but static allocations that take such characterisation into account tend to produce under-utilized platforms. It is, then, evident that optimal resource allocation algorithms cannot cope with this type of problem, and that lightweight heuristic solutions are needed. A comprehensive survey of the kinds of resource allocation heuristics that can cover different levels of dynamicity, while coping with the scale and complexity of high-density many-core platforms, is available in [15].

To cope with dynamism, a dynamic approach to resource management is the most obvious choice, aiming to dynamically learn and react to changes to the load characteristics and to the underlying computing platform. Linux has a strong momentum in the embedded software industry and has, in the past years, become the prevalent choice of operating system for new platforms. A paradigm, based on resource reservation, can endow applications with timing and throughput guarantees, independently of the good or malicious behavior of other applications, and can be employed across all system resources, including processor cycles, communication bandwidth, disk bandwidth, and storage.

The work presented here is the enhancement of previous work by the same authors [16,17] and the remaining of this paper is organized as follows. In the next section, we introduce the system model with the formal definitions for the network, communication protocol and scheduling behavior along with details of the orchestration process. Then, we describe the implementation of our system, and finally, we evaluate the results and conclude the paper.

2. System Model

We now proceed with the description of our system model by introducing formal definitions along with several considerations about its behavior. It is important to note that it is the programmer's responsibility to identify decomposable problems that can be used in this scenario. In a high level perspective, our system integrates the following features:

Data decomposition and assignment of data to nodes: Work is decomposed in several pieces, where the number of pieces is a function of the number of available nodes, and their size is proportional to each node's performance index.

Communication and failure management: There is a need to send data for processing to chosen nodes, to wait for the results and manage any eventual failures. Whenever a node fails, the work that was not processed is returned to the decomposition phase as a new instance of the process.

Mapping of results: Final result computation and its return to the application.

We use an orchestrator-based approach in order to achieve this. The details will be clarified in the following sections. We now proceed with some definitions and further explanations.

Definition 1 (Task)**.** *We define a task, t_i, as a λ function. By its nature, it will have no side effects and can be executed in parallel with other λ functions.*

In the remaining of this paper we will use the term task and lambda function for describing the same unit of execution and we use the term IoT device and node with the same meaning.

A device that needs to offload tasks to others can rely on a cluster of IoT devices for accomplishing this goal. We now define a cluster, which is the set of nodes currently available, meaning they are currently accepting tasks to execute.

Definition 2 (Cluster of IoT devices)**.** *Given an IoT device, we represent it by a node n_i. A cluster has a number of nodes, which can be variable during the execution of a computationally intensive application and is defined as $\mathcal{S} = \{n_1, \ldots, n_k\}$, where $k \geq 1$ and $n_i \in \mathcal{S}$ is one of the nodes currently available. The nodes can enter and leave the cluster at any time, as a result, for example, of a power failure (in case of leaving) or a new device is turned on (in case of entering).*

A cluster of nodes can be ordered from the more powerful to the less powerful members by evaluating their capabilities in terms of processing power and memory. Our option was to adopt a pragmatic approach, by implementing a simple heuristic function that relates clock speed, available CPU, number of cores, available RAM and available battery life. Details on how we get this data are described in the implementation section. We now define the device performance index.

Definition 3 (Device Performance Index)**.** *We define a function, $\mathcal{P}$, that given a node, n_i, its CPU speed, Cs_{n_i} (measured in Ghz), the number of cores, Cc_{n_i}, the available CPU capacity, Ca_{n_i} (measured in a number between 0 and 1), the available RAM, M_{n_i} (measured in Gigabytes), and the available battery, B_{n_i} (measured in a number number between 0 and 1), returns the value $\mathcal{P}(n_i)$, which is a numerical estimate for n_i performance based on the following formula:*

$$\mathcal{P} = \alpha * (Cs_{n_i} * Cc_{n_i} * Ca_{n_i}) + \beta * M_{n_i} + \delta * B_{n_i}$$

This is an easily computed value that, even if it is a relatively rough approximation, is, nevertheless, enough to distinguish each node's execution capacity without the burden of online benchmarking. It is also the programmer's responsibility to define adequate values for α, β and δ to produce an adequate value for his/her application.

Example 1. *Given a node, n_0, reporting the following data: $Cs_{n_0} = 1.4$, $Cc_{n_0} = 4$, $Ca_{n_0} = 0.6$, $M_{n_0} = 0.37$ and without battery information, and given $\alpha = \beta = 0.5$, the application of the formula results in:*

$$\mathcal{P} = 0.5 * (1.4 * 4 * 0.37) + 0.5 * 0.6 = 1.336$$

Example 2. *Given a node n_1 reporting the following data: $Cs_{n_1} = 1.5$, $Cc_{n_1} = 4$, $Ca_{n_1} = 0.15$, $M_{n_1} = 0.54$, the application of the formula results in:*

$$\mathcal{P} = 0.5 * (1.5 * 4 * 0.54) + 0.5 * 0.15 = 1.695$$

Knowing each node's performance index, we now define how to decompose the problem in order to distribute it in a balanced manner.

Definition 4 (Simple Problem Decomposition)**.** *Given a problem $\mathcal{D}$ and given a cluster of available nodes $\mathcal{S} = \{n_1, \ldots, n_k\}$, then the problem must be decomposable in k parts and defined as $\mathcal{D} = \{d_1, \ldots, d_k\}$ such that each part's computational cost is proportional to the assigned device performance index.*

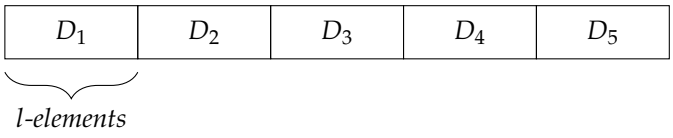
Example 3. *Given nodes $n_0, \ldots, n_5$ and a problem of summing 100,000 numbers, the calculated performance index, the percentage of the computational power each node represents and the assigned partition of the problem is presented in the following table:*

Node	***Performance Index ($\mathcal{P}_i$)***	***Percentage of System Power (p_i)***	***Assigned Partition***
n_0	*2.013*	*21%*	*21,000*
n_3	*1.965*	*20%*	*20,000*
n_1	*1.695*	*18%*	*18,000*
n_4	*1.472*	*15%*	*15,000*
n_2	*1.336*	*14%*	*14,000*
n_5	*1.125*	*12%*	*12,000*

A strict decomposition can be a bad solution if the computational cost of processing data is unevenly distributed, since a small interval of data can be harder to process than a larger one. The approach we purpose includes the option to split the work in a bounded number of parts that are processed sequentially by the cluster of nodes. We now define the enhanced problem decomposition.

Definition 5 (Enhanced Problem Decomposition)**.** *Given a problem, $\mathcal{D}$, and given a cluster of available nodes, $\mathcal{S} = \{n_1, \ldots, n_k\}$, then the problem must be decomposable in n parts and defined as $\mathcal{D} = \{D_1, \ldots, D_n\}$ and for each $D_i \in \mathcal{D}$, it is possible to decompose it further into k parts and defined as $D_i = \{d_{i1}, \ldots, d_{ik}\}$, such that each part's computational cost is proportional to the assigned device performance index. Thus, given a node, n_i, with a percentage of system power, p_i, then the size of the part, D_i, it will process is given by $p_i * sizeof(D_i)$.*

Example 4. *Given the Example 3, if we choose to have five partitions, then we get:*



Here, each node n_i will process $p_i * l$ elements of each D_i corresponding to:

Node	***Performance Index ($\mathcal{P}_i$)***	***Percentage of System Power (p_i)***	***Part D_k Size***
n_0	*2.013*	*21%*	*4200*
n_3	*1.965*	*20%*	*4000*
n_1	*1.695*	*18%*	*3600*
n_4	*1.472*	*15%*	*3000*
n_2	*1.336*	*14%*	*2800*
n_5	*1.125*	*12%*	*2400*

Given the previous definitions we can now define the assigned problem.

Definition 6 (Assigned Problem)**.** *Given a cluster of nodes $\mathcal{S} = \{n_1, \ldots, n_k\}$, where each node n_i has a performance index, $\mathcal{P}_i$, and the problem, $\mathcal{D}$, which is decomposed in k different parts, we define an assigned problem as a set of triples, $\mathcal{A}_P = \{(n_1, \mathcal{P}_1, d_1) \ldots (n_k, \mathcal{P}_k, d_k)\}$.*

Communication between nodes is done using asynchronous message passing. There is a permanent link between the node requesting the work and the nodes executing that work. When this link is broken, it signals a loss of communication and the node is removed from the list of available ones.

Definition 7 (Link set)**.** *Given a cluster of available nodes $\mathcal{S} = \{n_1, \ldots, n_k\}$ we define $\mathcal{L} = \{l_1, \ldots, l_k\}$ as the list of links to the nodes such that the connection to node n_k is done by link l_k.*

Failure during the execution of a task results in rescheduling the unfinished task to the closest available node in terms of performance index. More formally, we define task reassignment.

Definition 8 (Task Reassignment)**.** *Given a cluster of nodes, $\mathcal{S} = \{n_1, \ldots, n_k\}$, where each node, n_i, has a performance index, $\mathcal{P}_i$, the problem, $\mathcal{D}$, decomposed in k different parts proportional to each of the nodes and the assigned problem $\mathcal{A}_P = \{(n_1, \mathcal{P}_1, d_1) \ldots (n_k, \mathcal{P}_k, d_k)\}$. When a link, l_j, assigned to a node, n_j, such that $(n_j, \mathcal{P}_j, d_j) \in \mathcal{A}_P$ fails, then the task, d_j, is reassigned to the node, n_m, such that $(n_m, \mathcal{P}_m, d_m) \in \mathcal{A}_P \setminus (n_j, \mathcal{P}_j, d_j)$ and $\mathcal{P}_m \geq \mathcal{P}_n$ for any $(n_n, \mathcal{P}_n, d_n) \in \mathcal{A}_P \setminus (n_j, \mathcal{P}_j, d_j)$.*

The orchestrator plays a central role in the system. It is responsible for the coordination of the different participants, dealing with the details of each device and providing the programmer with an API that abstracts the use of the distributed system. Thus, its main features are:

- communication between all the participants;
- adding nodes to the cluster and removing nodes from the cluster;
- task distribution; and
- fault tolerance.

The orchestrator relies on the host that needs to offload work to other nodes. We now describe in more detail the concepts behind each of its features and other relevant details will be clarified in the implementation section.

Communication is done by message passing. All the different participants behave like actors [18]. All the messaging relies on the built-in features of the Erlang language that provide high-level approaches to message passing and code distribution, facilitating the whole process.

IoT devices can enter and leave the cluster at anytime. When they enter, they are available to accept tasks to execute. The node starts by sending a registration message to the orchestrator and, after acknowledgment, sends its score, which results from a performance index computation in the node. This will allow the orchestrator to rank that specific node within the cluster. Nodes can leave the cluster in two different scenarios: (i) when the orchestrator is shutdown; and (ii) when they stop, for example, due to power failure. In the first case, a message is sent from the orchestrator to the node, terminating the collaboration process. In the second one, the orchestrator detects the node's failure and removes it from the list of available nodes. Again, these features rely strongly on the built-in features of Erlang.

The API that the orchestrator provides accepts code and data, and returns the result of applying the code to the data. Its main goal is to distribute the data by node. It starts by splitting the data by the different nodes using their rank (given by the score obtained by the computed device performance index) to create partitions of data that each one will process.

This is illustrated in Figure 1, where we have an input to the orchestrator consisting of code and data, it distributes the code by the different nodes $\{N_1, N_2, \ldots, N_n\}$ and data $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, such that they can process it locally. For each N_i, the size of D_i varies accordingly with N_i's rank.

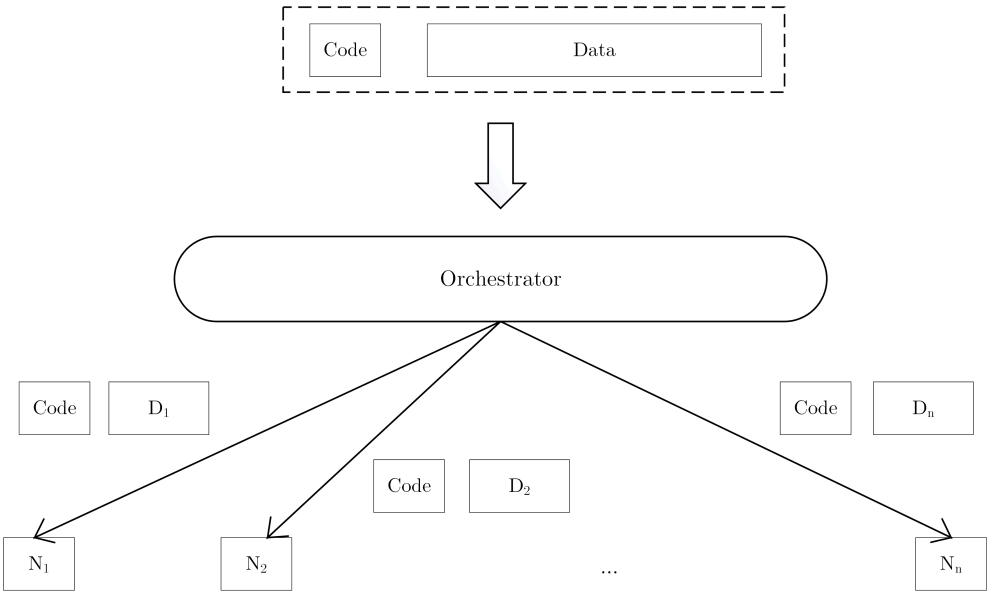


Figure 1. Orchestration process.

It is also the role of the orchestrator to provide fault-tolerance. Here, fault-tolerance consists in guaranteeing that all the data is processed. Since data is split among different nodes, failure in one or more than one node during execution results in losing the corresponding partial result. To guarantee the completion of the designated task, the orchestrator maintains a permanent link with each node in the cluster and detects any failure. In case of failure, the task given to that node is rescheduled for execution in the available node with highest performance index.

3. Data and Code Distribution Algorithm

We now present the core algorithms of this framework. Algorithm 1 describes how data is split and Algorithm 2 describes how data is distributed among the nodes. In Algorithm 1 we start by choosing from the simple problem decomposition of data (Definition 4) or the enhanced problem decomposition (Definition 5). In the first case, we split data by the number of available nodes proportionally to each node's performance index. On the other hand, if the enhanced problem decomposition is chosen, then an additional parameter (here described by the variable k) is provided, allowing a first split of the partition into k parts, then, for each of these parts, the data is split again, now in p parts (given p, the number of available nodes) with each of these parts with a size proportional to the nodes' performance indexes. This allows a more fine-grained distribution to the computational power of nodes with respect to the data being processed, which is particularly useful when the processing data has an uneven processing cost.

Having all the parts of data defined, we proceed with the distribution of the data and the supplied code for processing the data (described as $\mathcal{F}$) by the different nodes as described in Algorithm 2. The result is then stored. In case of a node being unable to complete a task, which translates into a broken link, the associated data must be processed by another node. The approach we use is to give it to the available node with highest performance to minimize further delays. In the case of several nodes failing, the process is repeated and the data is queued to the best available node. The algorithm hides most of the low-level technicalities which will be further discussed in the following sections.

Algorithm 1 Data Split

Let $\mathcal{D} := \{D_1, \ldots, D_p\}$ be the set of data which is divided in p parts.
Let $\mathcal{D}' := \{\}$ be an empty set of data pairs.

if Simple Problem Decomposition **then**
    $p := n$ and the size of partition D_i is adjusted to be proportional to p_i.
    $\mathcal{D}' := \mathcal{D}' \cup \{(D_1, n_1), \ldots, (D_p, n_p)\}$
end if
if Enhanced Problem Decomposition **then**
    A size k is provided, $p := k$ and
    for each $D_j \in \mathcal{D}$ **do**
        Split D_j in $\{d_{j1}, \ldots, d_{jp}\}$ where each d_{ji} is proportional to p_i.
        $\mathcal{D}' := \mathcal{D}' \cup \{(d_{j1}, n_1), \ldots, (d_{jp}, n_p)\}$
    end for
end if
return $\mathcal{D}'$

Algorithm 2 Data Distribution

Let $\mathcal{S} := \{n_1, \ldots, n_n\}$ be the set of available nodes in the cluster.
Let $\mathcal{L} := \{l_1, \ldots, l_n\}$ be the set of links to nodes in the cluster, where l_i is the link to node n_i.
Let $\mathcal{S}_\mathcal{P} := \{(n_1, p_1), \ldots, (n_n, p_n)\}$ be the set of pairs of available nodes in the cluster where each n_i is the node name and p_i is node's i performance index.
Let $\mathcal{D}'$ be the result of execution of the previous algorithm.
Let $\mathcal{F}$ be a function to process data in $\mathcal{D}$.
Let $\mathcal{R} := \{\}$ be an empty set of results of processing data in $\mathcal{D}$ by function $\mathcal{F}$.
Let *Success* := *False*

while Success = False **do**
    while $\mathcal{D}'$ has data **do**
        Remove (d_a, n_b) from $\mathcal{D}'$
        Submit data d_a for execution by n_b with code $\mathcal{F}$
    end while
    Wait until all nodes return a response and add them as a tuple (n_i, r_i) to $\mathcal{R}$, where r_i is the value returned by node n_i.
    if failed links exist **then**
        Add unprocessed requests (d_a, n_b) as (d_a, n_c) to $\mathcal{D}'$, where node b is replaced by the best one available, c.
    else
        Success := True
    end if
end while

4. Implementation

Although the idea is to have a general purpose solution for IoT devices, at this moment, we decided to focus on a specific type of hardware/software to develop a proof of concept with all the properties we believe that are relevant in this domain. Our nodes are all single-board computers, namely Raspberry Pi devices [19]. They all run a Linux distribution and an Erlang virtual machine.

Although single-board computers (SBC) are just one type of IoT device, they enjoy enormous popularity due to their high performance for their price range and the vast number of scenarios where they can be used [19,20]. It is possible to have several Raspberry Pi SBCs in the same area, each with a different purpose. With our framework we enable the optimization of devices that are, often, sitting idle.

4.1. IoT Node Implementation

The IoT node must have installed the Erlang VM to allow code transfer, execution and communication. Whenever an IoT device is available for collaboration, it searches for a registered orchestrator (in Erlang, registered processes are those that have a name associated with them) and uses the Erlang built-in instruction, $net\_adm : ping('orchestrator\_name')$, to register with the orchestrator. On success, the ping will add the IoT device to the list of known neighbors in the orchestrator process. Then, the device will compute its performance index using *sysbench* (https://github.com/akopytov/sysbench, accessed on 22 November 2021) and Linux's integrated *acpitools* and sends it to to the orchestrator. This allows the orchestrator to rank the node. From here on, the node is ready to be used as part of the cluster.

The code deployed to a node is initially minimal and consists of a simple process that executes code as instructed by received messages and is described in Listing 1.

Listing 1. IoT node main code.

```
task_executor()->
receive
{ From,execute,Mod,Fun,Param } ->
From ! { B,E,Mod:Fun(Param) },
task_executor();
_ ->
task_executor()
end .
```

The function, *task_executor*/0, waits for messages instructing the node to execute code (function *Fun* from module *Mod* with parameters *Param*) and the result of the execution is returned to the requesting node.

4.2. Orchestrator Implementation

The orchestrator is executed on the node that needs to offload data. This node, as all the others nodes that may form a cluster, runs an Erlang VM and the orchestrator is activated whenever it needs to offload work to others. After activation, the orchestrator is able to build a cluster of IoT devices and knows each one's performance score. Therefore, it can split tasks accordingly. In more detail, after knowing the list of available nodes $\mathcal{S}$ (nodes that successfully registered and sent their performance index), the first step is to create a ranked list. Given a list of nodes, $\mathcal{S} = \{n_1, \ldots, n_n\}$, they will be ordered in a list from the one with highest performance index to the one with lowest performance index. Their relative computational power will be used to compute the partition they must handle.

The orchestrator then proceeds with the transfer of code, $\mathcal{C}$, and data, $\mathcal{D}$, to nodes in the cluster. Erlang provides an easy way to do this. Given a module, *Mod*, on the device where the orchestrator is running, the Erlang instruction $c : nl(Mod)$ will transfer it to the nodes in $\mathcal{A}$. After this step, all the nodes have the same version of the code and data and know the partition they will work on, which is also sent by the orchestrator.

Please note that this approach may not escalate well when the problem being decomposed does not have an even distribution of work. It is also unable to split inter-dependent pieces of data. Nevertheless, these type of problems can also be handled by the orchestrator, but data will not be partitioned. Instead, all the application will be transferred to the node with the highest performance index.

After all the work has been distributed, the orchestrator waits for the results and handles failures. Waiting for results means it will wait for an answer from each node with the result of its particular execution. In case there is some failure, for example one node disconnects, it detects this because it relies on the underlying Erlang mechanism that generates a message whenever one element of $\mathcal{S}$ stops working. Thus, it is easy to know if one node stopped working and which part of the data it was processing. In this case, the orchestrator reschedules this exact execution to another node. The criteria implemented

is to wait until the end of the current execution and reschedule the uncompleted one to the highest ranked node.

The orchestrator node coordinates the offloading process and, typically, nodes can be both an orchestrator and a node executing tasks. The orchestrator Erlang function is succinctly described in Listing 2.

Listing 2. Orchestrator node main code.

```
orchestrator(Mod,Fun,DataSize,NPart) ->
NodesList = initialize_cluster(),
c:nl(Mod),
Parts_Node = distribute(NodesList,DataSize,NPart),
Results = execute(Mod,Fun,Parts_Node),
resechedule(Results,Parts_Node).
```

The function *orchestrator*/4 receives the name of the module, *Mod*, with the code and data that must be distributed, the main function processing the data, *Fun*, the size of the data being processed, *DataSize*, and the number of partitions, *NPart*, that should be used in the distribution of the work. Next, the function *initialize_cluster*/0 finds nodes in the local network area that are able to collaborate (execute the slave function described before), and adds them to the *NodesList*, establishing a link. Code and data are then distributed to the available nodes with the Erlang builtin function $c:nl/1$ and *distribute*/3 determines which parts of the data partition must be processed by which nodes. In case *NPart* is one, the process is a simple problem decomposition, if $Npart > 1$, then the process is an enhanced problem decomposition. The next step is to send the data for remote execution and gather all the results in the *Results* list. Finally, the function *resechedule*/2 compares the results obtained from nodes with the requests that were made. In case it detects unanswered requests (resulting from nodes failing during execution), tasks related with those requests are rescheduled to available nodes as described in the system model.

Example 5. *Given a module, primes, where a function, sum_primes*$(B, E, List)$*, returns the sum of the prime numbers in the interval from B to E in List and given a list of 100,000 numbers, summing all the primes in the list can be done using the cluster of IoT devices available, by submitting the following instruction to the orchestrator:*

```
orchestrator:process(primes,sum_primes,1000000,1)
```

By calling this function, all the features previously described are used to create a collaborative effort of handling the problem.

5. Evaluation

We carried several tests using a cluster of four devices connected wirelessly to the same WiFi router. These four IoT devices were in use with different main applications as described in Table 1.

Table 1. Cluster Setup.

Node	Device	CPU	Clock	RAM	Application
n_1	Raspberry Pi 3 B+	quad-core	1.4 GHz	1.0 GB	arcade machine
n_2	Raspberry Pi Zero W	single-core	1.0 GHz	0.5 GB	network add-blocker
n_3	Raspberry Pi 3 A+	quad-core	1.4 GHz	0.5 GB	wireless print server
n_4	Raspberry Pi Zero W	single-core	1.0 GHz	0.5 GB	no application

To evaluate the benefits of our framework, we have developed a battery of tests based on data-decomposable problems. Since results are consistent among the several tests, here, we present one of them, consisting of counting prime numbers in the interval [1, 50,000].

The interval we use in this test is $\mathcal{I} = \{1, \ldots, 50,000\}$ with a total of 5133 prime numbers. Note that the primes are not evenly distributed in this interval and higher ones are considerably more difficult to find than lower ones. We started with one device and added devices to the cluster measuring the gain in performance. Since the problem is not easy to break in balanced partitions we use the enhanced problem decomposition solution and break it in several partitions (1, 5, 10 and 20). Each of the partitions is then split by the available nodes accordingly with their reported performance index. We choose the node n_1 to be the node running the orchestrator although it can also execute tasks. The calculation of the primes on $\mathcal{I}_1$ took an average of 26.07 s. We don't get any advantage in using more than one partition with only one device since the implementation already uses multiple processes to optimize the use of the available cores of the device. The results of distributing data by the nodes are presented in Figure 2.

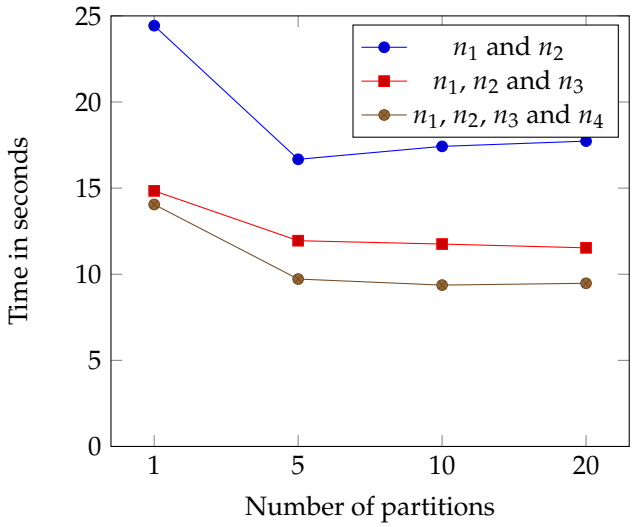


Figure 2. Distributing data by nodes.

Note that, if we add one device (n_1 and n_2), even with only one partition, the time needed to compute all the primes decreases from an average of 26.07 s to an average of 24.43 s. The performance increases as we divide chunks of work by the devices. With five partitions, we achieve the best result of an average 16.67 s. The increase in the number of partitions is not alone a factor of enhancement in performance, since, the more partitions we have, the more messages we need to exchange. When using devices n_1, n_2 and n_3, the performance increases considerably, which seems normal since n_3 is a powerful node in this context. Adding node n_4 also further increases the cluster's performance. In Table 2, one can see the percentage of gain in terms of execution time when adding, one, two and three devices to the cluster. The advantage is evident, although the gain when adding three devices when compared to two devices is marginal. One thing we notice is that problems have, generally, an ideal number of devices in the cluster to get the better trade-off between the size of the problem and the cluster setup and communication overhead.

Table 2. Percentage of gain by adding devices.

Devices Added	Gain When Compared to Single Device
1	55.8%
2	64.1%
3	65.2%

In Figure 3 we present the graphic detailing the gain of having one device (no devices added to help), with one, two and three devices added. These values are the ones for the configuration with best performance in each of the scenarios.

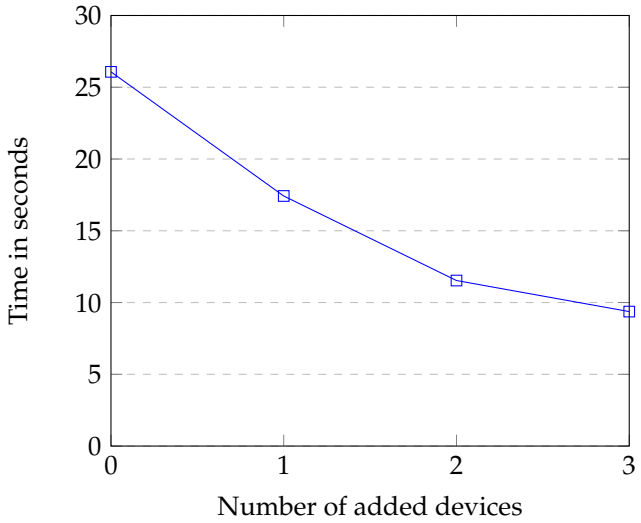


Figure 3. Result of adding devices to n_1.

We also experimented with failure in nodes and consequent rescheduling. The impact of such operation is highly dependent on the capacity of the node or of the nodes failing. Failing node n_3 has a considerable higher impact than failing node n_2, due to their different capacity and thus the amount of work that is distributed to them. Nevertheless, with a small number of failures, the cooperative distributed computation still has a better performance when compared to the single problem solving solution. In terms of time needed for the execution we conclude that given a set of nodes $\{n_1, \ldots, n_k\}$, the time, X, needed to decompose the service, the time, Y, to send code and data to a node, the time, W, to send the result back from a node and, t_{n_i}, the time that node n_i needs to processes its block, the total time, $\mathcal{T}$, needed for a distributed service execution can be determined by:

$$\mathcal{T} = X + k * Y + \sum_{i=1}^{k} \frac{t_{n_i}}{k} + k * W$$

From a general perspective and focusing only in the framework we developed and not the problems it may solve, we are able to draw some conclusions about the scalability of our framework. The division of work is done in two different approaches, one using a simplified split of data based on the performance index of the available nodes, and another based on the split in a given number of partitions and again, the split of each partition given the number of available nodes and their performance index. Both operations have low complexity since they have no relation with the size of the data being processed. In terms of communication and message passing the number of messages needed to setup the framework are equal to the number of available nodes and the number of messages needed to send and receive data are twice the number of partitions assigned to each node. Thus, the complexity of the whole framework setup and distribution of data and code is low. In the case of the enhanced problems assignment and depending on the data being processed, there is a compromise between the number of partitions given to each node and the time of the execution of the code over the data. Sometimes, depending on the data it may be faster to have less partitions, avoiding the communication overhead. General complexity of distributed systems has been previously studied in research such as [21–23].

6. Conclusions

Even though IoT devices are becoming more powerful, the available local resources cannot cope with the increasing computational requirements of resource-intensive applications that can be offered to a large range of end-users. This has created a new opportunity for task offloading, where computationally intensive tasks need to be offloaded to more resource powerful devices. Naturally, cloud computing is a well-tested infrastructure that can facilitate the task offloading. However, cloud computing, as a centralized and distant infrastructure, creates significant communication delays that cannot satisfy the requirements of the emerging delay-sensitive applications.

To this end, in this paper we presented a cooperative framework for IoT devices based in Single Board Computers and the Erlang programming language. The goal is to maximize the collaborative power of these devices with a minimal setup and interference on their main functions. By distributing the computational load across a set of heterogeneous IoT nodes, a cooperative environment enables the execution of more complex and resource-demanding services that otherwise would not be able to be executed on a stand-alone basis or would suffer from unacceptable performance. We intend to add more features to the framework and foresee the creation of a distributed solution for computation that uses available power of simple devices replacing larger systems.

Author Contributions: Conceptualization, J.C. and L.N.; methodology, J.C. and L.N.; software, J.C.; validation, L.N.; investigation, J.C. and L.N.; writing—original draft preparation, J.C.; writing—review and editing, J.C. and L.N.; visualization, J.C. and L.N.; supervision, J.C.; project administration, J.C. and L.N.; funding acquisition, J.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Artificial Intelligence and Computer Science Laboratory, University of Porto (LIACC), FCT/UID/CEC/0027/2020, funded by national funds through the FCT/MCTES (PIDDAC).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Cheng, C.; Lu, R.; Petzoldt, A.; Takagi, T. Securing the Internet of Things in a Quantum World. *IEEE Commun. Mag.* **2017**, *55*, 116–120. [CrossRef]
2. Nogueira, L.; Coelho, J. Self-organising Clusters in Edge Computing. In *Intelligent Systems Applications in Software Engineering*; Silhavy, R., Silhavy, P., Prokopova, Z., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 320–332.
3. Khan, M.A. A survey of computation offloading strategies for performance improvement of applications running on mobile devices. *J. Netw. Comput. Appl.* **2015**, *56*, 28–40. [CrossRef]
4. Kumar, S.; Tyagi, M.; Khanna, A.; Fore, V. A Survey of Mobile Computation Offloading: Applications, Approaches and Challenges. In Proceedings of the 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), Paris, France, 22–23 June 2018; pp. 51–58. [CrossRef]
5. Noor, T.H.; Zeadally, S.; Alfazi, A.; Sheng, Q.Z. Mobile cloud computing: Challenges and future research directions. *J. Netw. Comput. Appl.* **2018**, *115*, 70–85. [CrossRef]
6. Meurisch, C.; Gedeon, J.; Nguyen, T.A.B.; Kaup, F.; Muhlhauser, M. Decision Support for Computational Offloading by Probing Unknown Services. In Proceedings of the 2017 26th International Conference on Computer Communication and Networks (ICCCN), Vancouver, BC, Canada, 31 July–3 August 2017; pp. 1–9.
7. Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design*; Prentice Hall PTR: Hoboken, NJ, USA, 2005.
8. Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A Survey. *Comput. Netw.* **2010**, *54*, 2787–2805. [CrossRef]
9. Cesarini, F.; Vinoski, S. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016.
10. Terrell, R. *Concurrency in NET: Modern Patterns of Concurrent and Parallel Programming*, 1st ed.; Manning Publications Co.: Shelter Island, NY, USA, 2018.
11. Warburton, R. *Java 8 Lambdas: Pragmatic Functional Programming*, 1st ed.; O'Reilly Media, Inc.: Greenwich, CT, USA, 2014.
12. Hausenblas, M. *Serverless Ops*; O'Reilly Media, Inc.: Greenwich, CT, USA, 2016.

13. Armstrong, J. A History of Erlang. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III), San Diego, CA, USA, 9–10 June 2007; ACM: New York, NY, USA, 2007; pp. 6-1–6-26. [CrossRef]
14. Kopestenski, I.; Van Roy, P. Erlang as an Enabling Technology for Resilient General-Purpose Applications on Edge IoT Networks. Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang (Erlang 2019), Berlin, Germany, 18 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–12. [CrossRef]
15. Indrusiak, L.; Dziurzanski, P.; Singh, A. *Dynamic Resource Allocation in Embedded, High-Performance and Cloud Computing*; River Publishers: Delft, The Netherlands, 2016. [CrossRef]
16. Coelho, J.; Nogueira, L. Orchestration of Clusters of IoT Devices with Erlang. In *Software Engineering Perspectives in Intelligent Systems*; Silhavy, R., Silhavy, P., Prokopova, Z., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 585–594.
17. Coelho, J.; Nogueira, L. Collaborative Task Processing with Internet of Things (IoT) Clusters. In *Science and Technologies for Smart Cities*; Paiva, S., Lopes, S.I., Zitouni, R., Gupta, N., Lopes, S.F., Yonezawa, T., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 293–304.
18. Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*; MIT Press: Cambridge, MA, USA, 1986.
19. Johnston, S.J.; Basford, P.J.; Perkins, C.S.; Herry, H.; Tso, F.P.; Pezaros, D.; Mullins, R.D.; Yoneki, E.; Cox, S.J.; Singer, J. Commodity single board computer clusters and their applications. *Future Gener. Comput. Syst.* **2018**, *89*, 201–212. [CrossRef]
20. Jabbar, W.A.; Wei, C.W.; Azmi, N.A.A.M.; Haironnazli, N.A. An IoT Raspberry Pi-based parking management system for smart campus. *Internet Things* **2021**, *14*, 100387. [CrossRef]
21. Ranganathan, A.; Campbell, R.H. What is the complexity of a distributed computing system? *Complexity* **2007**, *12*, 37–45. [CrossRef]
22. Fraigniaud, P.; Korman, A.; Peleg, D. Towards a Complexity Theory for Local Distributed Computing. *J. ACM* **2013**, *60*, 1–26. [CrossRef]
23. Cabri, G.; Leonardi, L.; Quitadamo, R. Tackling Complexity of Distributed Systems: Towards an Integration of Service-Oriented Computing and Agent-Oriented Programming. In Proceedings of the 2008 International Multiconference on Computer Science and Information Technology, Wisla, Poland, 20–22 October 2008; pp. 9–15. [CrossRef]