Collaborative Task Processing with Internet of Things (IoT) Clusters

Jorge Coelho[1,2(✉)] and Luís Nogueira[1]

[1] School of Engineering (ISEP), Polytechnic Institute of Porto (IPP), Porto, Portugal
jmn@isep.ipp.pt
[2] LIACC Research Centre – University of Porto, Porto, Portugal

Abstract. We propose a framework for a collaborative processing of resource intensive services among Internet of Things (IoT) devices. Our goal is to optimize the use of this type of devices, particularly those being underutilized. The infrastructure bellow our framework is typically built with heterogenous appliances that have specific functions and the idea is to minimize the need for software updates and other changes, trying to use their spare resources with minimal interference. We base our solution in a pragmatic approach to task offloading based in the Erlang programming language.

Keywords: Edge computing · Computational offloading · IoT · Functional programming · Erlang

1 Introduction

Given the ubiquity of Internet of Things (IoT) devices and their strong proliferation [3] many opportunities appear in exploring their potentialities, namely their connectivity and computational power [11]. 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, 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 a more efficient execution [7,9,13]. These solutions rely on the concept of offloading to the cloud. However, due to the increase of 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 is not always a necessity if the available resources of these devices are wisely used.

S. Paiva et al. (Eds.): SmartCity360° 2020, LNICST 372, pp. 293–304, 2021.
https://doi.org/10.1007/978-3-030-76063-2_21

The study of scenarios where heterogeneous nodes with unknown resources are aggregated in a collaborative effort to achieve some goal was the subject of works such as [10] where some sort of data analysis is needed to estimate each node capacity and distribute work wisely.

Functional programming is an established approach to implement parallel and distributed systems [2]. The minimization of the need of a shared state enables code distribution and parallel processing which 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 [14,15] and is used in major cloud infrastructures such as the AWS Lambda [4]. In particular, Erlang [1] due to its simplicity and strong support for fault tolerant distributed programming, is seen as a promising language for IoT applications [8].

In this paper, we propose an Erlang-based framework for 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 to offload tasks and relying on a module that coordinates all the communication process and does a balanced scheduling of tasks based on their estimate computational cost and the device's availability in terms of computational power.

Distributing tasks implies knowing what nodes are available for this collaborating process, the amount of resources each node can offer and the approximate complexity of each task. These information will allow us to decide which node is the best one to process a given task. Note that we want to keep the changes at the node level minimal in order to not compromise its original function. Thus, we consider the use of resource reservation approaches [5,12] at the operating system level. A paradigm based on resource reservation can endow applications with timing and throughput guarantees independent of the behavior of other applications, and can be employed across all system resources including processor cycles, communication bandwidth, disk bandwidth, and storage. This way, IoT devices can cooperate and execute offloaded tasks while being able to establish the maximum amount of resources that can eventually be at use by the proposed framework.

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. Then, we describe the implementation of our system, and finally, we evaluate the results and conclude the paper.

2 System Model

Here we describe the system model of our framework 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. Our system is presented in a very simplified high level description in Fig. 1 and described next:

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 the need to send data to process to chosen nodes, wait for the results and manage their eventual failures. Whenever a node fails, the work that was not processed goes back to the decomposition phase as a new instance of the process.

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

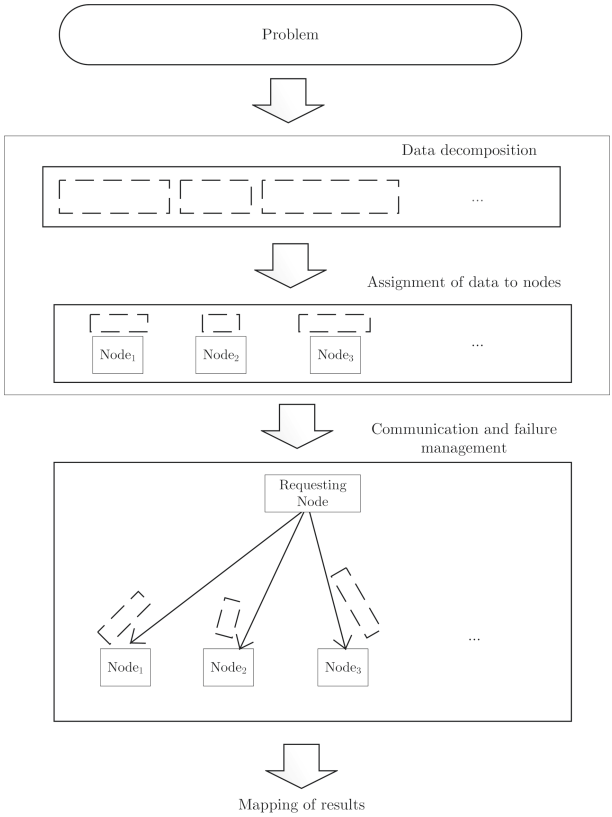


Fig. 1. Cooperative task execution overview

We now proceed with some definitions.

Definition 1 (Task). *We define a task t_i as a λ function. By 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 computational 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 anytime as result for example of power failure (in case of leaving) or a new device is turned on (in case of entering).*

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. Therefore, resource allocation is one of the most complex problems in large multi-processor and distributed systems, and in general it is considered NP-hard.

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.

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. A static allocation decided before deployment based on the (nearly) complete knowledge about the load and the platform, is not viable. 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 [5].

A cluster of nodes can be ordered from the more powerful to the less powerful 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 and available RAM. 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 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 remaining battery B_{n_i} (measured in a number between 0 and 1), returns the value $\mathcal{P}(n_i)$ that 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 easy to compute value that, even if it is a relatively rough approximation, is enough to distinguish node capacity without the burden of online benchmarking. It is also a programmer responsibility to define adequate values for the α, β and δ to produce 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.*

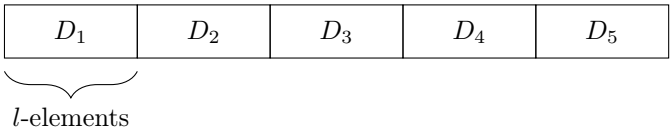
Example 3. Given nodes $n_0, \ldots, n_5$ and a problem of summing 100000 numbers, the calculated performance index, the percentage of the computational power each node represents and the assigned partition of the problem is presented in the following table:

Node	Performance Index ($\mathcal{P}_i$)	Percentage of system power (p_i)	Assigned partition
n_0	2.013	21%	21000
n_3	1.965	20%	20000
n_1	1.695	18%	18000
n_4	1.472	15%	15000
n_2	1.336	14%	14000
n_5	1.125	12%	12000

A strict decomposition can be a bad solution if the computational cost of processing data is unevenly distributed. 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 in 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 5 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 concepts, we 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* $(n_n, \mathcal{P}_n, d_n) \in \mathcal{A}_P \setminus (n_j, \mathcal{P}_j, d_j)$*.*

3 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 [6]. They all run a Linux distribution, an Erlang virtual machine and RPI-Monitor[1].

Although single board computers (SBC) are just one type of IoT devices, they enjoy enormous popularity due to the high performance for their price range and the vast number of scenarios where they can be used [6]. 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 many times sitting idle.

The computation of a node's performance index relies on the use of the RPI-Monitor utility, a general-purpose monitoring application that allows the extraction of several metrics from the Raspberry Pi. These metrics can be obtained by querying the device thought HTTP.

A node can be a slave (accepting work), a master (offloading work) or both. The code deployed to a slave node is initially minimal and consists in a simple process that processes execution messages and is described in Listing 1.1

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

Listing 1.1. Slave node main code

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.

[1] https://xavierberger.github.io/RPi-Monitor-docs/index.html.

The master node coordinates the offloading process and, typically, nodes can be both a master and a slave. The main master Erlang function is succinctly described in Listing 1.2.

```
master(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).
```

Listing 1.2. Master node main code

Function *master*/4 receives the name of the module (*Mod*) with the code and data that must be distributed, the main function processing the data *Fun*, size of the data being processed (*DataSize*) and the number of partitions (*NPart*) that should be used in the distribution of the work. Next, function *initialize_cluster*/0 finds nodes in the local network area that are to collaborate (execute the slave function described before), and adds them to the *NodesList* establishing a link. Code and data is then distributed by 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, 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.

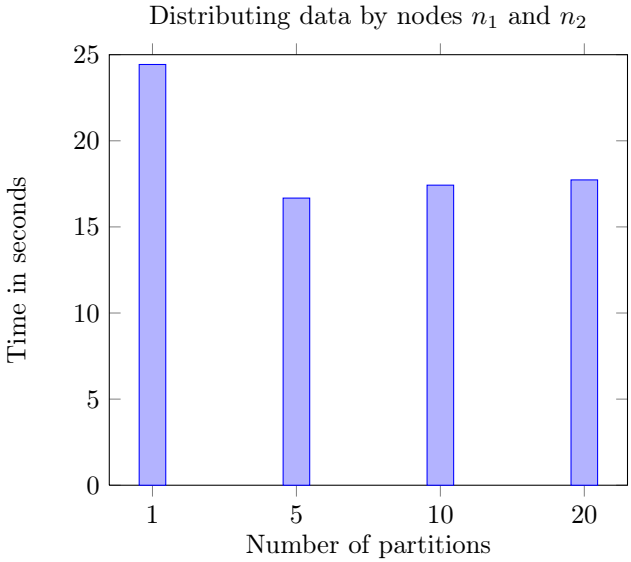
4 Evaluation

To evaluate the performance gain of our approach, we implemented an exhaustive set of benchmarks. For the sake of space, we present one benchmark based in a theoretically simple but still challenging problem, *i.e.* finding prime numbers in a given interval. We use a cluster of 4 IoT devices as described in Table 1.

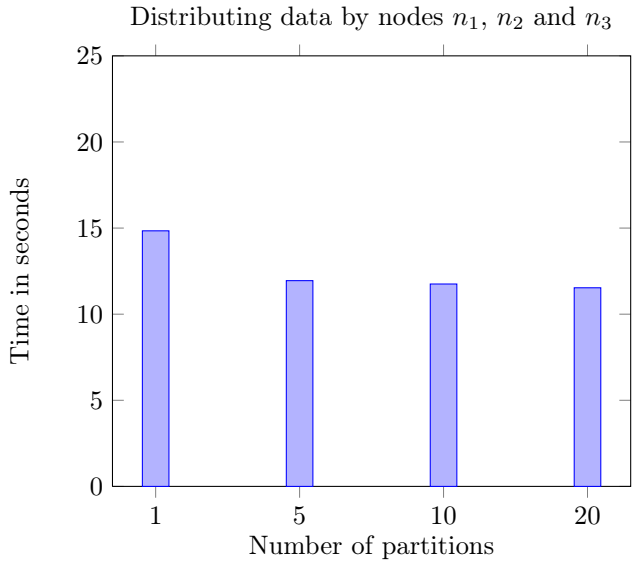
Table 1. Cluster setup

Node	Device	CPU	Clock	RAM
n_1	Raspberry Pi 3 B+	quad-core	1.4 GHz	1.0 GB
n_2	Raspberry Pi Zero W	single-core	1.0 GHz	0.5 GB
n_3	Raspberry Pi 3 B+	quad-core	1.4 GHz	1.0 GB
n_4	Raspberry Pi Zero W	single-core	1.0 GHz	0.5 GB

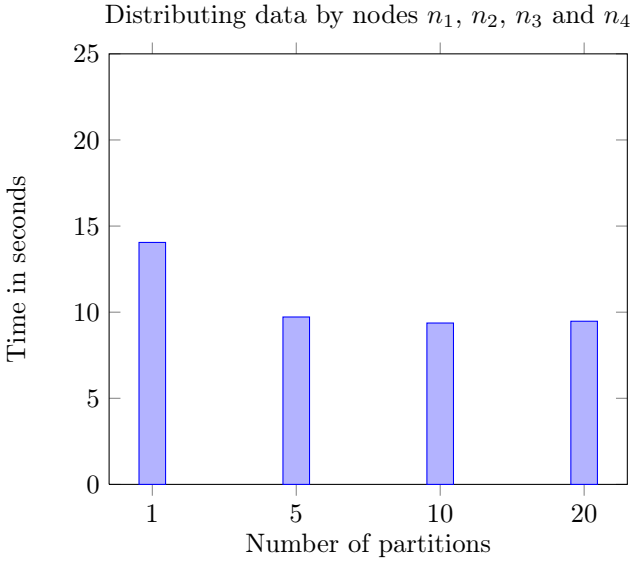
The interval we use in this test is $\mathcal{I} = \{1, \ldots, 50000\}$ with a total of 5133 prime numbers. Note that the primes are not evenly distributed in this interval and higher ones are considerably more difficult to find than lower ones. We started with one device and added devices to the cluster 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 master, although it also executes code as a slave. 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 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 two nodes (n_1 and n_2) are presented next:



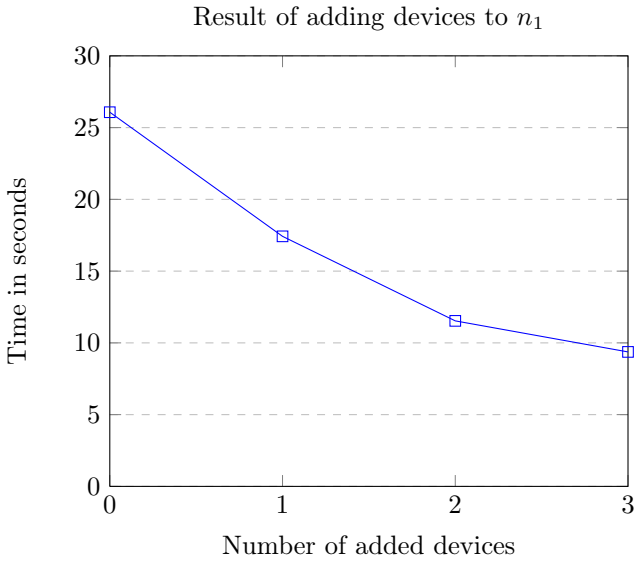
Even with only one partition, the time needed to compute all the primes decreases from an average of 26.07 s to an average of 24.43 s. The performance increases as we divide chunks of work by the devices. With 5 partitions, we achieve the best result of an average 16.67 s. The increase in the number of partitions is not alone a factor of enhancement in performance since the more partitions we have, the more messages we need to exchange. Next, we present the results of adding n_3 to the cluster. Here, the performance increases considerably, which seems normal since n_3 is a powerful node in this context.



Finally, we present the results of adding all the nodes:



As a summary, we present the next 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.



By adding one node (n_2) to the cluster, one gets an increase in performance of $\sim$ 33%. By adding two devices (n_2 and n_3), one gets an increase in performance of $\sim$ 55% and finally by adding three devices (n_2, n_3 and n_4), one gets an increase in performance of $\sim$ 64%. 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_4 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.

5 Conclusions

A cooperative execution of resource intensive services among heterogeneous IoT nodes is a promising solution to address the increasingly demanding requirements on resources and performance. In this paper, we presented a 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.

The obtained results make us believe that it is possible to use the spare computational power of each of these devices such that their cooperation enables the solution of computationally complex problems, which are difficult to solve in single devices with an acceptable 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.

Acknowledgments. This work was partially supported by LIACC (UIDB/00027/2020) through Programa de Financiamento Plurianual of FCT (Portuguese Foundation for Science and Technology).

References

1. Armstrong, J.: A history of erlang. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pp. 6-1–6-26. HOPL III, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1238844.1238850
2. Cesarini, F., Vinoski, S.: Designing for Scalability with Erlang/OTP: Implement Robust, 1st edn. Fault-Tolerant Systems. O'Reilly Media Inc., Sebastopol (2016)
3. Cheng, C., Lu, R., Petzoldt, A., Takagi, T.: Securing the internet of things in a quantum world. IEEE Commun. Mag. **55**(2), 116–120 (2017). https://doi.org/10.1109/MCOM.2017.1600522CM
4. Hausenblas, M.: Serverless Ops. O'Reilly Media Inc., Sebastopol (2016)
5. Indrusiak, L., Dziurzanski, P., Singh, A.: Dynamic Resource Allocation in Embedded. High-Performance and Cloud Computing. River Publishers, Gistrup (2016). https://doi.org/10.13052/rp-9788793519077. http://www.sciencedirect.com/science/article/pii/S0167739X18301833
6. Johnston, S.J., et al.: Commodity single boardcomputer clusters and their applications. Future Gener. Comput. Syst. **89**, 201–212 (2018)
7. Khan, M.A.: A survey of computation offloading strategies for performance improvement of applications running on mobile devices. J. Netw. Comput. Appl. **56**, 28–40 (2015). https://doi.org/10.1016/j.jnca.2015.05.018
8. Kopestenski, I., Van Roy, P.: Erlang as an enabling technology for resilient general-purpose applications on edge IoT networks. In: Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, pp. 1–12. Erlang 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3331542.3342567
9. Kumar, S., Tyagi, M., Khanna, A., Fore, V.: A survey of mobile computation offloading: applications, approaches and challenges. In: 2018 International Conference on Advances in Computing and Communication Engineering (ICACCE), pp. 51–58, June 2018. https://doi.org/10.1109/ICACCE.2018.8441740
10. Meurisch, C., Gedeon, J., Nguyen, T.A.B., Kaup, F., Muhlhauser, M.: Decision support for computational offloading by probing unknown services. In: 2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–9 (2017)
11. Nogueira, L., Coelho, J.: Self-organising clusters in edge computing. In: Silhavy, R., Silhavy, P., Prokopova, Z. (eds.) Intell. Syst. Appl. Softw. Eng., pp. 320–332. Springer International Publishing, Cham (2019)
12. Nogueira, L., Pinho, L.M.: A capacity sharing and stealing strategy for open real-time systems. J. Syst. Archit. **56**(4–6), 163–179 (2010). https://doi.org/10.1016/j.sysarc.2010.02.003
13. Noor, T.H., Zeadally, S., Alfazi, A., Sheng, Q.Z.: Mobile cloud computing: challenges and future research directions. J. Netw. Comput. Appl. **115**, 70–85 (2018). https://doi.org/10.1016/j.jnca.2018.04.018
14. Terrell, R.: Concurrency in.NET: Modern Patterns of Concurrent and Parallel Programming, 1st edn. Manning Publications Co., Greenwich (2018)
15. Warburton, R.: Java 8 Lambdas: Pragmatic Functional Programming. O'Reilly Media Inc., Sebastopol (2014)