Concurrent migration of containers in decentralized cloud computing network

Andrej Erjavec
89201090@student.upr.si
Faculty of Mathematics, Natural Sciences and Information Technologies,
University of Primorska
Glagoljaška 8
SI-6000 Koper, Slovenia

Aleksandar Tošić*
aleksandar.tosic@upr.si
Faculty of Mathematics, Natural Sciences and Information Technologies,
University of Primorska
Glagoljaška 8
SI-6000 Koper, Slovenia

ABSTRACT

In this paper we propose a new algorithm for decentralized application orchestration in Nion Network. The proposed algorithm improves on the existing implementation by performing multiple migrations between nodes such that it does not create race conditions. Moreover, we show experimental data testing various statistical measures of fitness in an effort to find the most suitable one. Our results show a significant improvement over the existing.

KEYWORDS

Decentralization, blockchain, edge computing, cloud computing, decentralized orchestration, application migration, containerization.

1 INTRODUCTION

The evolution of the internet and a growing number of its services started to reveal the shortcomings of centralized service architecture that has become unsuited to certain emerging trends [6]. Cloud computing, which is, like other internet services, highly centralized, is one of the major fields where the disadvantages of centralization are the most obvious [5]. Among these are low fault tolerance, a significant amount of processing load on data centers, and a difficult process of service migration between cloud infrastructure providers (vendor lock-in). Edge computing aims to address some of these challenges by distributing system resources as well as data across a multitude of nodes [3], but soon it was realized that edge devices are significantly underutilized in terms of system resource consumption and the coordination required for consistent system operation still relies on a centralized orchestrator. These findings have driven the development of a blockchain-based protocol for decentralized cloud computing, named Nion Network [4]. The protocol addresses multiple issues of existing solutions, the most significant change it delivers is support for fully decentralized orchestration. In addition, it aims to solve the issue of edge device heterogeneity by enabling their homogeneous operation that is independent of hardware configuration. The latter is achieved by packing applications into containers and performing migrations of containers during run-time. The migrations are validated by a consensus mechanism and are recorded on the blockchain in order to provide a verifiable and transparent log of system state changes.

*Also with InnoRenew CoE.

DOI: https://doi.org/10.26493/scores23.10

2 MOTIVATION

The migrations of containers in Nion Network are determined by a deterministic algorithm that produces a migration plan based on system resource statistics of nodes inside a network. These are collected and propagated to a block producer in each slot when a new block is created. The main goal of migrations is to evenly distribute a resource load across the network. However, the current implementation of the algorithm only supports up to one migration to be performed in each block, which significantly increases the time needed for the network to stabilize and hence reduces its overall performance. The operation of the algorithm is based on the concept of minimizing the load of most loaded nodes while maximizing the load of least loaded nodes to achieve an even distribution of system resources. Each migration can be represented as a vector containing three values:

$$\{containerToMigrate, sourceNode, destinationNode\}$$

The algorithm always takes the most loaded node as a source and the least loaded node as a destination. Among containers from the most loaded node, the one with the highest CPU usage is selected to be migrated. In addition, the impact of the proposed migration is calculated before the execution to evaluate whether the migration would improve the state of the system. As a measure of system stability, the difference between the most and the least loaded node is taken and compared before and after migration. For simplicity, the CPU load is expressed in percentage where we assume the value is normalized by hardware performance.

The migration is executed only in case the difference is lower after execution. In case the migration is performed, its goal is reached since the migration contributes to a more even distribution of resource consumption. However, when observing the possible scenarios, we noticed that the proposed migration might not be ideal since only one container is considered in the process of selection. There might exist another migration between the same nodes that would produce a lower difference once executed. Additionally, by not considering all possible containers from the most loaded node, there is a possibility of migration not being performed in the current slot due to a calculated negative impact on system stability, which leaves the system deprived of a possible improvement.

3 CONTRIBUTION

In the practical part of this study, we implement an algorithm that addresses the issues of the current implementation. The problem

```
Data: BlockData
Result: Migration plan
if !AppQueue.isEmpty() then
    while !AppQueue.isEmpty() do
        Min ⇐ FindMinLoadedNode(BlockData);
        Min.addApp(AppQueue.dequeue());
    end
else
    AppToMigrate ⇐ Max.MaxLoadApp;
    DeltaScore ⇐ (Max.score − Min.score);
    NextDeltaScore ⇐
     (Max.score − AppToMigrate.score) − (Min.score
     AppToMigrate.score);
end
if Math.abs(DeltaScore > NextDeltaScore) then
    Migrate(AppToMigrate, Min);
end
```

Algorithm 1: Deterministic migration algorithm

we are dealing with is similar to process scheduling with the goal of minimizing the total time required to complete all tasks. In this context, optimization algorithms like Longest-processing-time-first (LPT) and Shortest-processing-time-first (SPT) are often used [2]. However, these algorithms are designed for systems where the number and difficulty of all tasks are known prior to execution making them unsuitable for a dynamic network like Nion where the load distribution is constantly changing with new applications entering the network in each slot.

3.1 Implementation

The proposed solution is designed to compute a migration plan based on the current state of the network. To provide a separate testing environment for new solutions, we implemented a simplified simulation of the Nion protocol, which tries to mimic the dynamic nature of the real network to ensure similar conditions. The implementation is executed in three steps where each aims to improve a different aspect of the algorithm, while the main focus is reducing the time needed for the network to stabilize.

3.1.1 Improvement 1: Enabling multiple migrations per block. First, we focus on solving the most significant limitation of the algorithm while keeping the existing concept for container selection. As a measure of system stability, the difference between the most and the least loaded node is taken as in the case of the original algorithm. Multiple migrations per block are enabled by building a migration plan inside a loop, which iterates as long as the migration proposed in the current iteration still improves the system stability. We also observed that once the local minimum of a difference is reached, at the same time the global minimum for that slot is reached as well which allows us to break the loop and return the produced migration plan.

3.1.2 Improvement 2: Improved procedure of container selection. The previous change to the algorithm enables multiple migrations per block but still leaves room for improvement. In this step, we address the problem of non-ideal migrations that can slow down the process of load balancing between nodes or miss a chance for stability improvement. The goal of this step is to consider all containers from the most loaded node in the process of selection, not only the one with the highest CPU usage. Instead of directly selecting the most loaded container, We first evaluate the impact of migration for each of these containers and select the one that has the greatest impact on system stability when migrated. As in the previous improvement, the migrations are being added to the plan as long as the migration in the current iteration produces a lower difference compared to the previous iteration.

3.1.3 Improvement 3: Use of standard deviation as a measure of system stability. Since the problem involves evenly distributing system resources, the use of standard deviation as a measure of system stability seems reasonable. With this approach, network stability is obtained by calculating the standard deviation of CPU usage across all containers on the network using the following formula:

$$stdDev = \sqrt{\frac{1}{|N|} * \sum_{n \in N} (n_{cpu} \overline{N}_{cpu})^2} \tag{1}$$

In the previous versions of an algorithm, the execution of migration is decided based on comparing the difference between the most and the least loaded node in the current and the previous iteration. However, these nodes are most likely to be different once a migration is performed which makes such a comparison method inappropriate and may result in sooner than desired termination of the algorithm. The use of standard deviation enables us to evaluate the impact of individual migrations not only on nodes involved in migration but on the network as a whole. The migration plan is still generated inside a loop with the difference in standard deviation now taking the role of a measure. Similar to previous cases the loop continued as long as the standard deviation was lower than the one in the previous iteration. In the previous cases, we were able to evaluate the impact of migration before the execution since only the involved nodes were considered. In the case of standard deviation, however, the same can not be achieved since the calculation of standard deviation involves all nodes and containers on the network. It is therefore not possible to make an evaluation before the migration is executed. For this reason, the migration has to be simulated prior to calculation in order to assess its impact on the network. An important note is that the integration of such an algorithm into an actual network would require the block producer node to have the ability to perform a simulation of migration. That can be achieved by implementing a system able of making a copy of the network state based on received resource statistics and using this model as a simulation environment.

This step also completes the final version of our solution (Algorithm 2).

4 RESULTS

In this section, we present the results obtained during the testing of improved versions of the migration algorithm. The tests are done for each of the mentioned improvements to compare its performance against previous implementations. Each test is performed based on average and worst-case scenarios. In the average case, a

```
   Data: BlockData
   Result: Migration plan[]
 1 MigrationPlan ⇐ [];
 2 DstCandidates ⇐ BlockData;
 3 StdDev ⇐ GetStdDev();
 4 StdDevPrev ⇐ StdDev;
 5 while StdDev <= StdDevPrev & DstCandidates.size > 0
   do
 6     Max ⇐ FindMaxLoadedNode(BlockData);
 7     Min ⇐ FindMinLoadedNode(DstCandidates);
 8     AppToMigrate ⇐ Max.MaxLoadedApp;
 9     Containers ⇐ Max.getContainers();
10     Containers.sort(CPU);
11     LoadDelta ⇐ Max.cpu − Min.cpu;
12     for Container : Containers do
13         NextDeltaScore ⇐
            (Max.cpu − AppToMigrate.cpu) − (Min.cpu
            AppToMigrate.cpu);
14         if NextLoadDelta > Delta then
15             AppToMigrate ⇐ Container;
16             break;
17         end
18         LoadDelta ⇐ NextLoadDelta;
19     end
20     SimulateMigration(AppToMigrate, Max, Min);
21     DstCandidates.remove(Min);
22     StdDevPrev ⇐ StdDev;
23     StdDev ⇐ GetStdDev();
24     if StdDev < StdDevPrev then
25         Migration ⇐ {AppToMigrate, Min, Max};
26         MigrationPlan.add(Migration);
27     end
28 end
```

Algorithm 2: Final version of migration algorithm

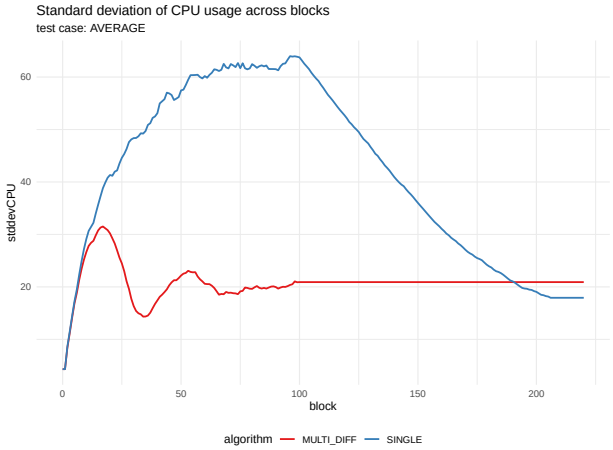

Figure 1: Comparison of performance between single and multiple migrations per block

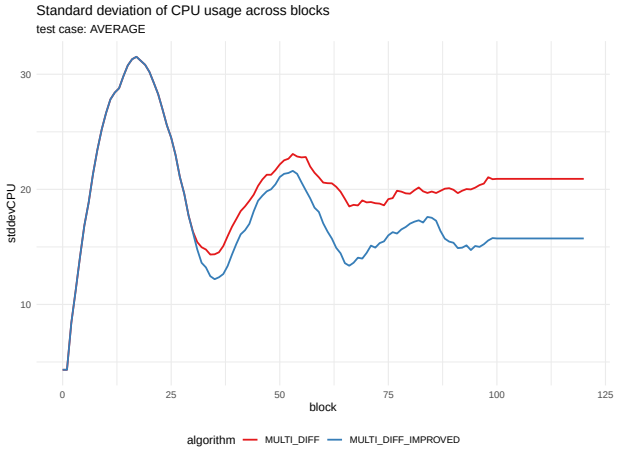

Figure 2: The effect of improved container selection on standard deviation

fixed number of containers is added to a randomly selected node in each iteration while in the worst case, the containers are always added to the same node. When a desired number of containers on the network is reached, the adding stops. We take the standard deviation of CPU load across the network as a measure of performance for all tests. The testing is performed on a simulated network with 1000 containers and 256 nodes.

After the first improvement, which enabled multiple migrations per block, we see significant improvement in terms of time required for stabilization (Figure 1). While it took around 200 blocks for the original algorithm to complete the stabilization, the new version required around 100 blocks, which makes it reach the minimum standard deviation almost two times faster. As shown in the figure, the standard deviation starts to decline much sooner when performing multiple migrations per block. However because of rapid stabilization in the early phases of the process when new blocks are still entering the network, the standard deviation is not in constant decline. When a stable-enough state is reached, new migrations can not improve the stability for the next sequence of blocks making the standard deviation rise again. The rising ends when enough new containers enter the network and an algorithm is again able to produce migrations that start improving the stability.

The next version of the algorithm where we implemented an improved routine for container selection shows an improved performance when compared to the initial multi-migration-per-block algorithm (Figure 2). Since the selection is such that the proposed migration has the largest impact on the stability, we are able to reach a lower standard deviation as in the previous case when a more naive approach for selection was used.

Changing the method for measuring stability with the use of standard deviation as a measure further contributed to reaching better stability of the network (Figure 3). The cause for the improvement is widening the scope on which the new algorithm is able

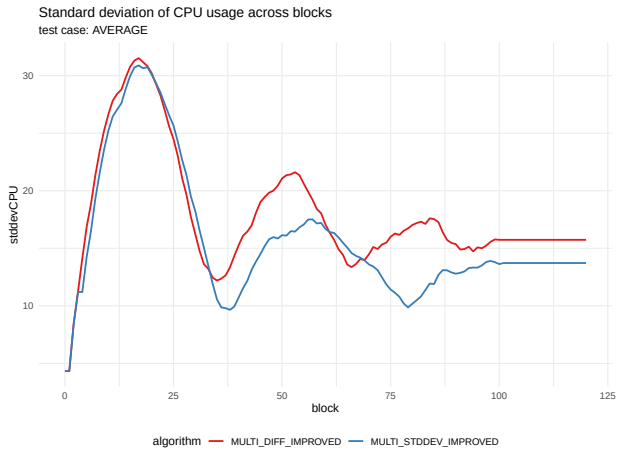

Figure 3: The effect of improved container selection on standard deviation

to evaluate the impact of migrations. This contributes to a higher number of migrations per block as we show in Table 1 resulting in a lower standard deviation reached.

Another important indicator, that can be used to assess the performance of our algorithm, is the number of migrations per block an algorithm is able to produce. We observe that the last improvement where the standard deviation is used in combination with improved container selection produces the highest number of migrations.

Table 1: Migrations per block based on algorithm

	number of migrations per block		
algorithm	min	mean	max
SINGLE	1	1	1
MULTI DIFF	1	2.79	9
MULTI DIFF IMPROVED	1	2.75	9
MULTI STDDEV	1	2.66	10
MULTI STDDEV IMPROVED	1	4.60	18

4.1 Quality of proposed solution

In the end, we want to measure the quality of our proposed solution. In this case, the algorithms for process scheduling represent a good reference for comparison. Even though their usage is unsuitable for our use case, the fact they provide an optimal distribution of resources enables us to make an evaluation of how close our solution is to being optimal. We took the Longest processing time algorithm (LPT) [1] as a reference since it provided the finest results. We tested the final version of our algorithm and collected the results of 20 tests where a new simulation network was generated in each test to provide variability in conditions. We present the results in Table 2.

As expected there is a gap in performance between algorithms, but considering the different use cases these algorithms are intended for, we conclude that the outcome generated by our solution is acceptable.

Table 2: Comparison of our algorithm with LPT

	std. dev. of CPU usage (%)	
case/algorithm	our implementation	LPT
average case	13.85	2.31
worst case	16.09	4.07

5 CONCLUSION

In this paper, we presented the main operational concept of the Nion protocol and highlighted the main drawbacks of its migration algorithm. We extended our findings by proposing an improved version of the algorithm that helps to achieve faster network stabilization times. Our solution enables multiple migrations per block and improves the concept of selecting the container to be migrated.

In the future, we could continue our work on improving the algorithm by implementing more advanced optimization techniques to further improve its performance and tune its operation to more specific requirements. One possible step in the evolution of our algorithm would be the use of multi-criteria optimization where multiple parameters are considered as opposed to the current solution where decisions are made based on CPU usage only. By setting priorities on the parameters we would get an algorithm that is more flexible and adjustable to current network needs and features.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the European Commission for funding the InnoRenew CoE project (H2020 Grant Agreement #739574) and the PHArA-ON project (H2020 Grant Agreement #857188) and the SRC-EDIH project (DIGITAL-2021-EDIH-01 call, project number: 101083351) and the Republic of Slovenia (Investment funding of the Republic of Slovenia and the European Union of the European Regional Development Fund), as well as the Slovenian Research Agency (ARRS), for supporting project number J2-2504.

REFERENCES

[1] Asep Anwar, Didit Damur Rochman, and Rendiyatna Ferdian. 2021. Parallel Machine Scheduling with Shortest Processing Time (SPT) and Longest Processing Time (LPT) TO Minimize MAKESPAN at PT. ABC. *Geographical Education (RIGEO)* 11, 6 (2021), 403–407.

[2] Jun-Ho Lee and Hoon Jang. 2019. Uniform Parallel Machine Scheduling with Dedicated Machines, Job Splitting and Setup Resources. *Sustainability* 11, 24 (2019). https://doi.org/10.3390/su11247137

[3] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[4] Aleksandar Tošić, Jernej Vičič, Michael Burnard, and Michael Mrissa. 2022. A Blockchain-based Edge Computing Architecture for the Internet of Things. (2022). https://doi.org/10.20944/preprints202111.0489.v2

[5] Magnus Westerlund and Nane Kratzke. 2018. Towards Distributed Clouds: A Review About the Evolution of Centralized Cloud Computing, Distributed Ledger Technologies, and A Foresight on Unifying Opportunities and Security Implications. In *2018 International Conference on High Performance Computing Simulation (HPCS)*. 655–663. https://doi.org/10.1109/HPCS.2018.00108

[6] Wenli Yang, Erfan Aghasian, Saurabh Garg, David Herbert, Leandro Disiuta, and Byeong Kang. 2019. A Survey on Blockchain-Based Internet Service Architecture: Requirements, Challenges, Trends, and Future. *IEEE Access* 7 (2019), 75845–75872. https://doi.org/10.1109/ACCESS.2019.2917562