Data Consistency in Mobile Collaborative Networks Based on the Drop Computing Paradigm

Vlăduţ-Constantin Tăbuşcă, Radu-Ioan Ciobanu, Ciprian Dobre
Faculty of Automatic Control and Computers
University Politehnica of Bucharest
Bucharest, Romania
vladuttabusca@gmail.com, {radu.ciobanu, ciprian.dobre}@cs.pub.ro

***Abstract*—In the past years, the use of smartphones and mobile devices on the widest scale has led to an evolution of existing mobile networks. This has generated different routing and dissemination algorithms compared to the existing ones for classic wired networks. Therefore, we introduced the Drop Computing paradigm, which proposes the concept of decentralized computing over multilayered networks, combining cloud and wireless technologies over a social crowd formed between mobile and edge devices which will receive every data or computation request. The problem addressed in this paper is related to the consistency of data received from other peers, as there may be situations where it has been intentionally corrupted or has suffered inevitable changes on its path. Our main contributions include a rating mechanism of the nodes and a waiting mechanism for more versions of the same information in order to establish the correctness of the received data. Using simulations, we show that the proposed solutions increase the consistency level up to 100% in some situations. The simulation of the network interactions is based both on a synthetic mobility model, as well as on a real-life mobility trace.**

***Index Terms*—mobile, social, opportunistic, consistency, edge, fog, cloud**

I. Introduction

Due to the advent of the Internet of Things and the proliferation of mobile devices that need to communicate (with each other and with central servers), classic constructs such as cloud computing can no longer keep up. For this reason, there is a tendency nowadays for networks to scale horizontally, towards edge and fog computing. On top of this, we have previously proposed the Drop Computing paradigm [1], which adds an extra layer composed of mobile devices in proximity. Thus, instead of going directly to the cloud or to an edge node, mobile nodes first query their nearby neighbors in an opportunistic fashion [2]. This offers an alternative for avoiding the high latency and cost of cloud computing, by utilizing the nodes nearby for serving and offloading a device's requests.

Based on this, we tackle the problem of data consistency in Drop Computing. Corrupted data (be it intentional or unintentional because of external factors such as malfunctioning sensors) leads to an unwanted behavior when utilizing mobile networks, so the solution we propose here attempts to offer mechanisms for ensuring consistency in Drop Computing. We aim to offer mobile nodes a higher confidence level in the information obtained and tasks computed in the surrounding network.

Thus, we propose solutions for ensuring data consistency in Drop Computing-based networks, where mobile nodes generate tasks and ask neighboring devices to help solve them. Our solutions include using a rating mechanism, as well as a system for expecting a certain number of task versions before making decisions regarding the correctness of the information received. These mechanisms are further improved by various limitations regarding the path a task takes through the Drop Computing network, such as the number of different executors, the maximum frequency of nodes visited, the familiarity of nodes, etc. Furthermore, we also implement a detection and correction mechanism for corrupted bits of data based on Hamming codes.

Through simulations, we show that we are able to increase the percentage of uncorrupted tasks that are received by a node, leading to values as high as 100% (i.e., no corrupted tasks are delivered to the application level). Our proposed mechanisms are able to correctly detect malfunctioning or malicious nodes and exclude them for communication. Thus, even in situations where the majority of the versions of a task present in the network at one point are corrupted, our proposed solution is able to correctly send the valid version to the owner's application level. The main challenge is to achieve acceptable delays when getting the result of a task to the application level, in order for as many tasks as possible to be successfully computed.

The remainder of this paper is structured as follows. In Sect. II, we discuss the main components of Drop Computing and the way it functions, and then we look at several solutions for ensuring data consistency in mobile cloud networks. We propose our solution in Sect. III and evaluate it in Sect. IV. Finally, we present our conclusions and future work in Sect. V.

II. Related Work

This section offers a quick introduction into the most important elements that the Drop Computing paradigm is based on (opportunistic networks and mobile edge computing),

This research is supported by University Politehnica of Bucharest, through the "Excellence Research Grants" program, UPB - GEX 2017, identifier UPB-GEX2017, ctr. no. AU 11.17.02/2017. It is also supported by NETIO project Tel-MonAer, as well as projects SPERO (PN-III-P2-2.1-SOL-2016-03-0046, 3Sol/2017) and ROBIN (PN-III-P1-1.2-PCCDI-2017-0734).

and then addresses data consistency in mobile networks, which is the main topic of this paper.

A. From Opportunistic Networks to Drop Computing

Opportunistic networks (ONs) are a sub-type of delay-tolerant networks (DTNs) and an extension of mobile ad-hoc networks (MANETS) [3]. The main difference between ONs and other types of mobile networks is the fact that they are based on the high mobility of the nodes involved, generating extremely dynamic routes between nodes. Moreover, because most connections are not stable and occur at irregular moments, most of the time there is no complete path between a sender and a receiver. Opportunistic networks are based on the store-carry-and-forward paradigm [4], which means that, when a node wants to send a message, the first step is to store it. Then, the message is carried around the network until a suitable next hop is found (or the destination). Upon a contact with a potential next hop, the message is forwarded, and this happens until the message reaches its final destination.

Mobile edge computing networks [5] propose scaling communication horizontally, as opposed to doing so vertically (towards the cloud infrastructure), when the cloud model is not feasible anymore. This paradigm offers an extremely distributed computing environment, used both for developing applications and services, and also for storing and processing data much closer to mobile devices. A mobile edge computing application can be divided into multiple sub-components that are spread independently in the network. This way, a part of the requests can be answered by a locally-formed cloud at the edge of the network, or by the central cloud.

Based on these two paradigms, Drop Computing proposes decentralizing computing over multi-layered networks, by combining cloud and wireless technologies over a social crowd composed of mobile and edge devices [1]. Thus, instead of directing every data or computation request towards the cloud, Drop Computing uses the devices in proximity for a quicker and more efficient access. The need for this paradigm comes from the insufficiency of the classic cloud model, which becomes unsuitable in the fast ascension of the Internet of Things. When a large number of small devices communicate with each other, they all need to send requests to the cloud, wait for the replies, and then process them, even if the devices are in close proximity to one another. By employing opportunistic communication, Drop Computing is able to extend the network and the cloud horizontally, adding an extra layer below the edge.

B. Data Consistency in Mobile Cloud Networks

Most of the communication paradigms used in the cloud assume that there is a connection at all times between a device and the owner/carrier of the information it requests. However, this might not always be true, so alternative methods of ensuring communication need to be found, especially in critical situations such as natural disasters or military actions, when the existing infrastructure might be damaged or unavailable. Techniques for ensuring data availability through replication thus need to be used, because data ends up being spread at multiple locations, reducing the response times and congestion.

In mobile networks, node mobility creates partitioning quite often, so organizing data to ensure consistency becomes a crucial problem. In [6], various consistency conditions of data replicated in MANET are presented, by classifying consistency levels based on the requirements of applications. The authors assume the existence of an environment where each device can access data from other nodes, thus replicating information in their own memory. The network is divided into multiple regions and the consistency is performed on a per-region basis. There are two types of devices in the proposed network: some have special characteristics and are called proxies (having unlimited memory and knowing how replicas are spread in the network), while others are regular nodes that only know their own replicas. The attributes that characterize consistency levels are global consistency, location-based consistency, time-based consistency, or device-based consistency. The authors show that hit rates and traffic loads are much better if a local consistency protocol is used, instead of a global consistency solution. However, such a solution is not feasible in Drop Computing, because nodes are not always connected to each other, so paths between two peers that need to communicate would not always be available.

In [7], distributed data replication techniques are proposed, in order to balance the compromise between delayed request responses and data availability. The One-to-One Optimization (OTOO) method uses each mobile node to collaborate with its neighbor and decide what information to store. Thus, each node computes a value of the frequency of combined access to a piece of data, and the stores this data based on the computed value. This method is somewhat limited by the fact that nodes are only able to communicate with one-hop neighbors, thus limiting their options. In Drop Computing, nodes communicate opportunistically, so this problem is averted by allowing peer collaboration across Wi-Fi or Bluetooth range boundaries.

III. Proposed Solution

In this section, we present the proposed solution for ensuring data consistency in Drop Computing.

A. Data Corruption

In our approach, we assume that data corruption can occur in two situations: either immediately after a task has been executed (when the executor's version gets corrupted), or when information about completed tasks is exchanged (when the corrupted version is the one belonging to the node that is receiving the completed task). Out of the two situations, it can be observed that the first one leads to the propagation of a corrupted task version to all the nodes encountered by the executor, whereas the second scenario leads to the corruption of the task only on certain paths from the executor to the task's owner. Thus, when corruption occurs at the executor, all the versions that it spreads in the network will be corrupted, since a corrupted task cannot be corrected. The only chance in this

scenario is the possibility that a task has multiple executors in the network that do not in turn corrupt the result, allowing the task to reach its owner on a path where its integrity is not affected.

B. Storing the Path of a Task

When a node sends a task to be solved in the network by other peers, and the result arrives back to its owner, it is important to know the path that it took, in order to correctly identify the nodes that have corrupted the data. The task circulates in the network in two forms: first it is unexecuted, and then as a completed task from its executor towards its owner, through the encountered nodes. Thus, there are many possibilities of transferring a task between two nodes (the same task can be transferred either executed or unexecuted, or it can be exchanged multiple times between the same two nodes, since nodes are able to delete tasks from their own memory when forwarding it to other devices). These transfer possibilities have been identified by simulating node mobility and analyzing real-life data collected from interactions between multiple devices.

In order to store the nodes that a task's version that arrives at its owner has passed through, we use a list that contains each path taken by each task at any point in time. Adding a next destination to this list should find a unique path taken by that task until the current time moment. This is based on the fact that, when a node deletes a task from its memory (generally when transferring it to another device), the corresponding path is deleted from the list, whereas a node that currently has a task at the same point in time (be it executed or not) does not download it anymore.

C. The Characteristics of Expected Versions

To improve the chances of a task being correctly received by its owner, we have implemented a waiting system for multiple versions of the same task prior to making a decision regarding the correctness of the data received. Thus, a task's results will not be sent to the application level until a certain number of versions arrive, which are used to select the most popular one.

Aside from this condition, we propose improving the solution by specifying that a certain percentage of executors of all versions of a task need to be different, while also adding the restriction that a minimum number of final relay nodes per task should be expected. These restrictions only make sense if the number of expected task versions is higher than 1, and can help increase consistency because of the following:

- by accepting task versions executed by different nodes, the chances that the information is not corrupt increase, regardless of the way data is corrupted
- by accepting task versions from different nodes, we allow the task to have a more diverse path from executor to owner, which is useful in the scenario where tasks are corrupted after they are computed.

The expected versions mechanism is presented in Alg. 1. When all expected versions are received, the majority version is computed, which is considered the correct version that will be accepted by the owner. In case two or more different versions have the same number of occurrences and thus there is no majority (lines 8-11), a decision cannot be taken and a new version is required, which will be used to form a majority. By going over the list of received information, a node decides whether there is a difference between its own unaltered data and the information received (lines 13-17). If the majority of the versions have been corrupted (lines 19-23), then the node has no choice but to accept corrupted data. However, this is something that we wish to avoid by implementing the rating system described in the next subsection.

Algorithm 1 Expected versions mechanism.

```
1:  L_data - list of received versions
2:  Task_id - received task id
3:  T_c - number of corrupted tasks
4:  N_versions - number of waiting versions
5:  Task_data - unmodified task data
6:
7:  Majority = percent of most frequent data from L_data
8:  if Majority is equal to 50% then
9:      increase number of waiting versions for Task_id by 1
10:     return
11: end if
12:
13: for all data D from list L_data do
14:     if detect collisions between D and Task_data then
15:         increase T_c by 1
16:     end if
17: end for
18:
19: if Majority > 50% and 2 × T_c > N_versions then
20:     corrupted version is accepted
21: else
22:     uncorrupted version is accepted
23: end if
```

D. Node Rating System

In order to classify the encountered nodes by the correctness of the tasks they spread in the network, we implemented a rating system using historical information. In a mobile network, certain nodes are more predisposed to data corruptions than others, due to external factors, but also intentionally. Thus, in order to prevent a high spread of corrupted data, it is important to posses knowledge about nodes with high chances of data corruption and to avoid them when possible.

Since we are dealing with a decentralized network, there cannot be a single central rating value for each node that can be accessed by anybody at any time. Instead, each node has its own list of ratings for all the other peers it encounters, which is updated upon a contact with another node. Furthermore, gossiping is used to obtain information about nodes not yet encountered, which is performed when nodes exchange information about completed or uncompleted tasks. This way, nodes have a more informed view of the network, and are able to avoid corrupted nodes even when they have not been encountered before.

Decreasing a node's rating is done after all expected versions of a task are received at the owner. After establishing the majority version, all others versions are marked as corrupt. The paths of the corrupted versions are analyzed and versions exchanged at each step are compared, in order to see which

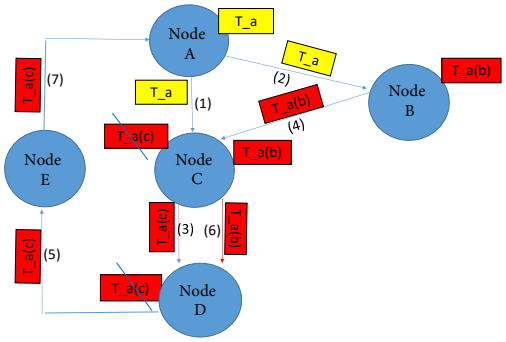


Fig. 1: Task exchange example.

exchange was at fault for the corrupted version. When such as exchange is found, the sending node's rating is decreased by the task owner, which then starts to gossip the new rating value upon each contact with other peers.

Extra care needs to be taken when deciding which node on the path has corrupted a certain task. For this reason, we add a timestamp to any transaction, in order to create correct paths that can easily be analyzed. Figure 1 shows a node transfer scenario where the timestamp is paramount to making correct decisions. Node A is the owner of task T_a and spreads it in the network for execution, the yellow color of the task showing that it has not been executed yet. At time moment 1, the task is delivered to node C, and at moment 2 to node B. Then, the task is solved by node C (which is what the red color specifies), which then sends it to node D at time moment 3 and deletes it from its own memory. Node B also computes the task and delivers it to node C at time moment 4. Since node C had previously deleted the task at moment 3, it will now receive the version computed by B. This shows that node C sees two versions of the task at two separate moments of time (and, as shown in Fig. 1, this is also true for node D). In this situation, if node B corrupts the task and sends a corrupted version to node C, if timestamps are not employed, the owner of the task (node A) will incorrectly assume that node C is the one that corrupted the task, since it had definitely received a correct version from the owner.

The entire node rating mechanism, which is executed whenever a node exchanges tasks with other peers, is presented in Alg. 2. The algorithm returns *false* if the encountered node should not be trusted (i.e., is not socially connected to the current node, or has a rating below 75, as shown at lines 19-23 and 26-30, respectively), and *true* otherwise. Since nodes might corrupt data because of a hardware or software problem and not necessarily out of maliciousness, the proposed algorithm allows such nodes the possibility of increasing their rating through good behavior. This is done using a timer per node, which is decreased every time the rating algorithm is run, as shown on line 8 in Alg. 2. When the timer expires, if the node has a promising rating (higher than 50%), the rating value is increased with 20% and the timer is reset (lines 12-15), thus

Algorithm 2 Node rating mechanism.

```
1:  dcNodeId - encountered node id
2:  NMap_R - map between encountered nodes id and rating values
3:  TMap_R - map between node id and timer value
4:  T_R - initial timer associated with the rating value
5:  Social - social network of the current node
6:
7:  if NMap_R contains node dcNodeId then
8:      decrease rating timer by 1 and add the new value in TMap_R
9:      R_value = value of the key dcNodeId stored in NMap_R
10:     T_value = timer value from TMap_R
11:
12:     if R_value is between 50 and 100 and T_value <= 0 then
13:         put in NMap_R minimum value between 100 and R_value+20
14:         put in TMap_R initial timer value T_R
15:     end if
16:
17:     R_value = new value from NMap_R
18:
19:     if R_value >= 75 then
20:         return true
21:     else
22:         return false
23:     end if
24: end if
25:
26: if Social contains dcNodeId then
27:     return true
28: else
29:     return false
30: end if
```

reincluding the node in the task computation and dissemination process. The node is then considered trustworthy as long as its rating stays above 75%.

E. Hamming Codes

Hamming codes are part of a family of error correction and detection codes that are able to detect two errors and correct one. In order to implement such a system, parity bits need to be added next to the actual data bits. These represent redundant information that contributes to correctly recovering the initial data even when it is corrupted. The overhead added by the parity bits reaches ideal (i.e., small) values when the quantity of data grows. Thus, we have added a Hamming code-based mechanism into our proposed solution. In this situation, simulating data corruptions is different, since a single error would always lead to a correctness percentage of 100%. Therefore, we still follow the corruption scenarios presented in Sect. III-A, but the number of corrupted bits is chosen randomly between 1, 2, and 3, so that errors are not always correctable.

IV. Evaluation

This section presents an evaluation of the proposed solution, showing the setup of the experiments and the results obtained.

A. Setup

We implemented and tested our solution using the MobEmu simulator[1] [8], which is able to run routing and dissemination solutions in mobile opportunistic networks. Drop Computing was already implemented in MobEmu, so we simply had to add our consistency mechanisms on top of the existing implementation.

[1]Available at https://github.com/raduciobanu/mobemu.

The first set of experiments was realized using the HCMM model [9], which simulates the behavior and the interactions between multiple nodes. This model includes both a behavior based on the membership of a specific social network and the attraction to some particular physical locations. For this scenario, we created the following simulation: 30 mobile nodes, split into 5 different communities, 6 hours of interactions, with 5 of the nodes being travelers that can move between communities. The physical space was simulated as a 1000x1000-meter grid, and the speed of the nodes was set to vary between 1.25 and 1.5 m/s. Finally, the transmission radius of the nodes was set to 10 meters. Along with this synthetic model, we also used a real-life mobility trace collected at our faculty, called UPB 2011 [10].

The proposed implementation was evaluated using 3 metrics, which allowed us to make measurements on multiple data consistency and corruption scenarios. These metrics were:

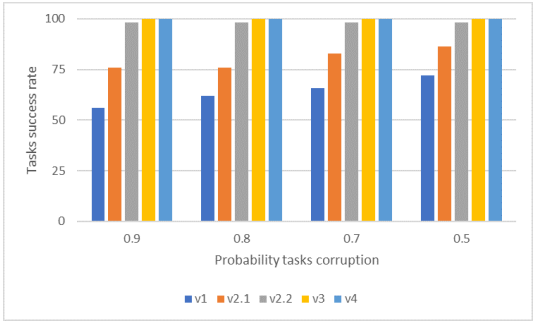
- the percentage of correct tasks received at their destinations, which were executed by other nodes in the network
- the number of the tasks executed by other nodes (this number varies according to the number of versions of the same task which must be expected before making a decision)
- the average processing latency for every task executed by other nodes (the time elapsed from the generation of the task until the moment when the owner accepts its execution).

The measured values were analyzed both quantitatively and qualitatively in the following situations (also presented in Table I:

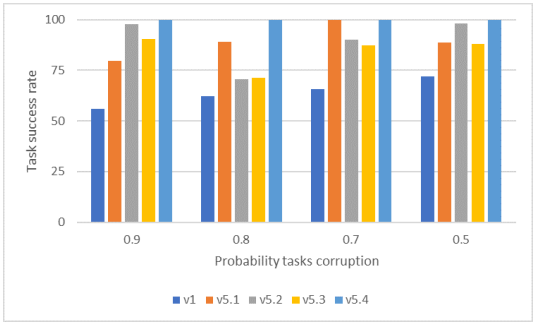
- the simulation of data corruption, taking into account both scenarios described in Subsect. III-A, without using any of the consistency mechanisms
- the variation of the number of expected versions (3 or 4, with the mention that, if there two versions with the same frequency, the node waits for another one for that task)
- the existence or absence of the rating mechanism (in the affirmative case varying the value of the rating's benchmark)
- varying the percentage of different executors depending on the number of desired versions
- using Hamming correction and detection codes (varying the number of permitted corruptions of the same task).

Version	Mechanisms	Observation
v1	Default Drop Computing	data can be corrupted any time
v2.1	v1 + 3 expected versions	-
v2.2	v1 + 4 expected versions	extra version expected if majority is 50%
v3	v1 + 2 expected versions + rating	2 experimentally proven to be the best value
v4	v2.1 + at most 2/3 versions executed by the same node	-
v5.1	v2.1 + Hamming	-
v5.2	v2.2 + Hamming	-
v5.3	v1 + Hamming	-
v5.4	v3 + Hamming	no multiple versions expected

TABLE I: Testing scenarios.



(a)



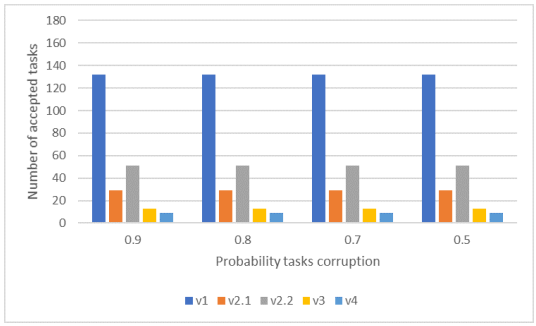
(b)

Fig. 2: Task success rate - HCMM (for more information about the testing scenarios, please see Table I).

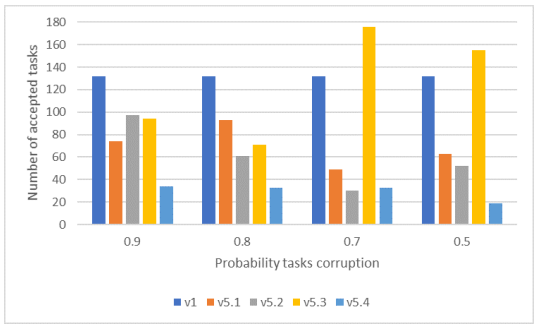
B. Results

Figure 2 shows the percentage of tasks correctly executed and transferred by the network nodes to the owner. In Fig. 2a, it can be observed that the standard version *v1* (where no corruption detection mechanisms are used) has a correctness percentage noticeably lower than all versions implemented in this paper. Some of our proposed techniques provide a fairness percentage of accepted tasks as high as 100%. Along with this perfect percentage, we also observed a considerable reduction in corrupted versions in the network by using the rating mechanism. Thus, from 857 corrupted tasks in the standard version, there were 15 corruptions in the *v3* scenario, suggesting that an efficient filtering of the corrupted nodes was produced, the data no longer being predisposed to corruption due to the avoidance at critical moments of nodes with low ratings.

Figure 2b shows all scenarios that use Hamming correction and detection codes, and it is important to observe how the percentage of tasks correctly received does not increase proportionally with the decrease of the probability of corrupted nodes. This behavior is due to the fact that, at the time of corruption, a random number between 1 to 3 is chosen, with the possibility that the proportion of corruption with more than one bit is higher even if fewer corruptions were recorded.

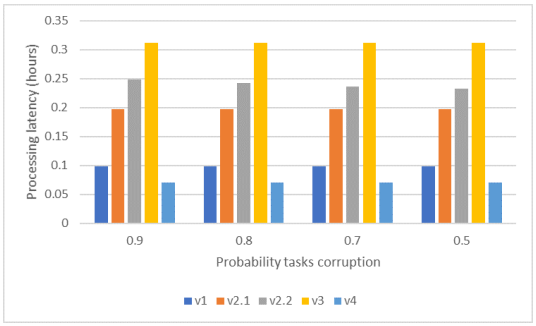


(a)

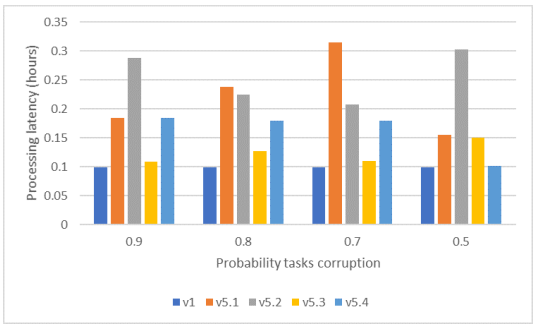


(b)

Fig. 3: The number of executed tasks in the network - HCMM.



(a)



(b)

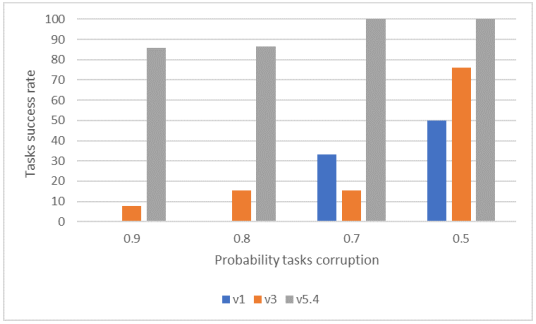
Fig. 4: Processing latency - HCMM.

However, it can be seen that in all scenarios shown in Fig. 2, a better share of correct results is recorded than in the standard version, *v1*, which confirms the improvements brought about by our solution.

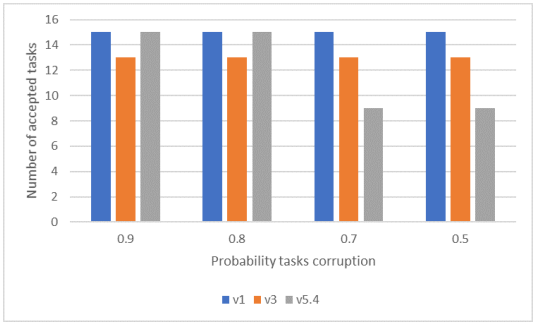
The lower value of the number of executed tasks in the network is confirmed by Fig. 3. The charts show that our solution provides a degree of reliability regarding the correctness of information at the cost of a lower number of tasks solved. This is due both to node filtering and to the fact that these simulations were made with the same amount of waiting time of a node for any task. Thus, with the increase of the waiting versions, chances for a node to execute its own tasks using only its own resources are also increased.

Figure 4 shows how the average processing latency for each given task executed in the network increases, but there are cases where its value is similar or even lower (e.g., *v4* or *v5.3*). *v4* is the most incontestable example demonstrating the improvements made, because it produces both a lower latency and a percentage of 100% of correct task acceptations, as highlighted in Fig. 2a. The existence of these situations shows that the entire network was efficiently covered, tasks being shared through nodes that actually had more chances to meet the task's owner, despite the decrease of transfer possibilities. Increased processing latency is somewhat inevitable when multiple versions are expected, but it can be seen that the percentage of duration increase is not proportional to the percentage of increase in the number of expected versions.

The results obtained for the UPB2011 trace are highlighted in Fig. 5. Due to the fact that only 20 mobile nodes were used



(a)



(b)

Fig. 5: The variation of correct tasks received - UPB 2011.

in this simulation, the percentage of nodes that are predisposed to corruption (30%) is higher than in the HCMM simulation (26%), all simulations being made with the assumption that 1 in 3 nodes can corrupt information. Thus, for a high

probability of corruption, less than 0.9, there was an absolutely 0% of correctly received tasks. Taking into account that this simulation has a small number of node connections, scenarios with multiple waiting versions or different executors could not be used. However, using the rating mechanism and Hamming detection and correction codes, an improvement of up to 80% of correctly received tasks is obtained. In Fig. 5a, at probability 0.7, it can be seen that the percentage of corrupt accepted tasks after applying the rating mechanism is lower than that obtained in the standard version. This is an isolated case due to the decreased rating of a node that has not yet been corrupted. Therefore, due to the lower number of connections between nodes, the current node fails to increase its rating above the reference value, not having enough opportunities to transfer the correct information it holds.

To conclude, the main advantage of this implementation is the high percentage of correct tasks received by the owner, thus suggesting a successful filtering of corrupt nodes, accomplished by optimized exchanges of useful messages. The improvements that we intend to make to this implementation are related to the decrease of processing latency and the increase of the number of executed tasks in the network. We want to achieve this by an optimal distinction of the nodes that are predisposed to corruption, both in order to avoid them, and also to learn the moments in which these corruptions occur.

V. Conclusions and Future Work

In this paper, we have proposed mechanisms for ensuring data consistency in Drop Computing, where devices have tasks that need solving and offload them to other devices in their proximity. The results have shown that, through setting the appropriate restrictions, the proposed solution can satisfy the requirements of a network with regard to a desired trust level. For example, when the tasks being computed in the network contain sensitive information, using the proposed rating mechanism can lead to a task correctness as high as 100%. This method exhibits a decrease in the number of messages and tasks exchanged through the network, but not necessarily proportional to the increase in corruption probability. This is caused by the fact that multiple nodes will reach a small rating value and be considered corrupt, which will lead to any interactions with these nodes being avoided by other peers, until the trust level is increased again. On the other hand, using the proposed waiting mechanism, combined with various limitations regarding the path a task takes or the node that executes it, leads to a high level of tasks computed by the other devices in the network. This scenario leads to a higher latency when compared to the default scenario, but also improves the probability of receiving uncorrupted data, which is the desired behavior.

For future work, we plan on improving the rating mechanism and attempt to find solutions for increasing the hit rate and lower the processing latency and overhead. Furthermore, we also want to increase node altruism by integrating new reward mechanisms for each node that executes and disseminates computing tasks. Aside from using the Hamming code, we also wish to implement a Reed-Solomon code [11], which has the advantage of detecting corrupted bits based on the number of parity bits added in the payload.

References

[1] R.-I. Ciobanu, C. Negru, F. Pop, C. Dobre, C. X. Mavromoustakis, and G. Mastorakis, "Drop computing: Ad-hoc dynamic collaborative computing," *Future Generation Computer Systems*, 2017.

[2] R.-I. Ciobanu, C. Dobre, D. G. Reina, and S. L. Toral, "A dynamic data routing solution for opportunistic networks," in *Telecommunications (ConTEL), 2017 14th International Conference on*. IEEE, 2017, pp. 83–90.

[3] R. Ciobanu, R. Marin, C. Dobre, V. Cristea, C. X. Mavromoustakis, and G. Mastorakis, "Opportunistic dissemination using context-based data aggregation over interest spaces," in *2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015*, 2015, pp. 1219–1225. [Online]. Available: https://doi.org/10.1109/ICC.2015.7248489

[4] R. Ciobanu, C. Dobre, and V. Cristea, "Reducing congestion for routing algorithms in opportunistic networks with socially-aware node behavior prediction," in *27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013, Barcelona, Spain, March 25-28, 2013*, 2013, pp. 554–561. [Online]. Available: https://doi.org/10.1109/AINA.2013.63

[5] Y. Yu, "Mobile edge computing towards 5g: Vision, recent progress, and open challenges," *China Communications*, vol. 13, no. Supplement2, pp. 89–99, N 2016.

[6] T. Hara and S. K. Madria, "Consistency management strategies for data replication in mobile ad hoc networks," *IEEE Transactions on Mobile Computing*, vol. 8, no. 7, pp. 950–967, 2009.

[7] P. Nithiyalakshmi and V. U. Kumar, "Data consistency for cooperative caching in mobile environments," *International Journal of Science and Research (IJSR)*, vol. 3, no. 1, 2014.

[8] R.-I. Ciobanu, R.-C. Marin, and C. Dobre, *MobEmu: A Framework to Support Decentralized Ad-Hoc Networking*. Cham: Springer International Publishing, 2018, pp. 87–119. [Online]. Available: https://doi.org/10.1007/978-3-319-73767-6_6

[9] C. Boldrini and A. Passarella, "Hcmm: Modelling spatial and temporal properties of human mobility driven by users social relationships," *Computer Communications*, vol. 33, no. 9, pp. 1056–1074, 2010.

[10] R. I. Ciobanu, C. Dobre, and V. Cristea, "Social aspects to support opportunistic networks in an academic environment," in *International Conference on Ad-Hoc Networks and Wireless*. Springer, 2012, pp. 69–82.

[11] A. R. de Araujo Zanella and L. C. P. Albini, "A reed-solomon based method to improve message delivery in delay tolerant networks," *International Journal of Wireless Information Networks*, vol. 24, no. 4, pp. 444–453, 2017.