Robust Intra-Slice Migration in Fog Computing

Atefeh Talebian, Alvin Valera, Jyoti Sahni, and Winston K.G. Seah
School of Engineering and Computer Science Victoria University of Wellington
Wellington, New Zealand
{atefeh.talebian, alvin.valera, jyoti.sahni, winston.seah}@ecs.vuw.ac.nz

***Abstract*—Low latency is critical to applications such as control of unmanned aerial vehicles. Such latency-sensitive services can be hosted closer to the user at the fog layer which can reduce overall latency through the reduction of transmission time and network congestion. To keep the latency low for mobile users connected to services deployed at the fog, these services need to be constantly migrated to follow the users. Unlike the cloud nodes, fog nodes are less reliable and are therefore subject to higher failure rate. In this paper, we propose an enhancement to the post-copy live migration algorithm to make it robust against failure. Simulation results show that robust migration reduces total migration time between 10-26% and downtime between 2-23% compared to non-robust migration. Furthermore, when the bandwidth to the backup node is lower, robust migration provides further improvement in both metrics.**

***Index Terms*—Post-copy migration, Fog computing, Network Slicing, Low latency, Reliability, Robust migration**

I. Introduction

Latency is a critical performance metric in cutting edge and futuristic Internet of Things (IoT) applications and services. Services that require real-time response such as command and control of *Unmanned Aerial Vehicles* (UAV) require low latency [1] since high and excessive latency fluctuations could cause catastrophic failure. To achieve consistently low latency, latency-sensitive services should be hosted physically closer to the user at the fog layer instead of the cloud layer. The physical proximity to the user brings resources (e.g., storage and computation) closer to the users, thereby reducing overall latency due to reduction in transmission time and network congestion.

To support service migration, there is a need to perform efficient resource allocation within the fog layer, to ensure that the service has the necessary resources across several fog nodes in a multi-tenant network. Fortunately, this problem is addressed using network slicing [2]. Using this technique, a slice satisfying the *Quality of Experience* (QoE) requirements can be allocated to the service [3].

As can be seen in Fig. 1, in scenarios such as UAVs where users are mobile, when such users host their services at fog nodes, there is an unavoidable need to move the service to follow the physical trajectory of the users [4]. Otherwise, this will result in the service being farther away from users, thereby increasing the latency or even losing the connection and therefore the service. Moving the service from one fog node to another fog node is essentially a migration problem.

Migration in general is one of the key techniques for resource management in fog computing [5]. It can be categorized

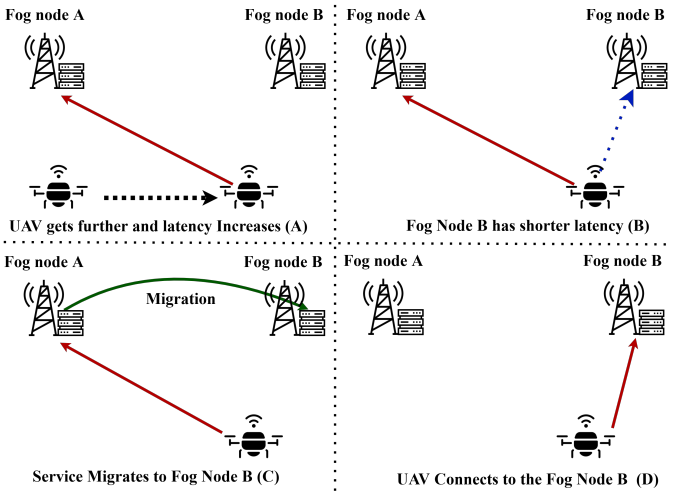

Fig. 1. Mobility Induced Migration

into either *cold* or *live* migration, and can be applied to either a *Virtual Machine* (VM) or a *container* that is hosting a service[1]. In cold migration, the hosted service is paused during the entire migration whereas in live migration, the hosted service is allowed to run and only paused for a short time. The time that the service is paused is called *downtime*, and it adversely affects the latency or response time of users that rely on the service. To achieve low latency during the migration it is preferable to use live migration rather than cold migration. This is due to the long downtime of cold migration which makes it unsuitable for migrating latency-sensitive services.

Among several live migration techniques, post-copy [6] is known to achieve better performance compared to other techniques not only because of its shorter migration time but more significantly due to its lower migration traffic and faster handover [7]. In post-copy, all memory pages are sent to the destination node only once and the execution state handover from the source to the destination node occurs at the early part of the migration process. These characteristics make post-copy migration suitable for low-latency sensitive services that serve mobile users.

One major problem of post-copy migration is robustness [8] which is becoming a concern in fog computing due to the lower reliability of fog nodes [1]. If the destination node fails

[1]Without loss of generality, we assume that a VM or container is hosting one service but in practice, they can host more than one service.

during the migration process[2], the service will be aborted, and even if recovery is possible, the recovered state of the service is stale. This happens because in post-copy, only the destination has the most up-to-date processor state. This is due to the fact that the execution state is handed over immediately (after it receives the processor state from the source). However, it does not have all the memory pages yet as it requests for these pages only when a page fault occurs. Thus, the failure of the destination node would result in the service losing the most current processor state and the updated memory pages. The consequences of stale state may result in bad user experience (for instance, in the case of live streaming) or in certain cases, they can be life-threatening or catastrophic (for instance, in case of the service controlling UAVs and self-driving cars). Even if the destination node sends updated memory pages and processor state to the source node [9] because of the mobility, the user may have lost the connection with the source node or suffer from long latency.

This paper will therefore address the robustness problem of post-copy live migration while preserving its low latency characteristic. Key to our algorithm is the inclusion of a back-up node that can take over in case of the destination node failure. The remaining part of this paper is organized as follows. Section II discusses the related work. Section III outlines the system model and algorithm design. Next, the simulation and results are provided in Section IV. Finally, Section V concludes this paper.

II. Related Work

In this section we review the work that focuses on post-copy migration. While live migration has lower downtime, in some cases, it may still negatively affect the user's QoE. During the migration process, page faults may happen frequently, increasing the total migration time and delay which will adversely affect low latency services. In this case the destination node will send a page fault request to the source node and will wait to receive it. These page faults increase the total migration time and delay during migration which will adversely affect low-latency services.

Several techniques have proposed to reduce the number of page faults during the post-copy. During the migration, page faults can be divided into two types, those caused by accessing a not-yet-received page and those caused by accessing an already received page by destination node. Shan et al. [10] designed a page table assistant in Xen which is able to indicate whether a VM status is in migration. With this technique Xen can distinguish whether a page fault happens because the page is not received or already received. Then the destination node will not send the page fault to the source node for the pages that have not been fault because of the migration. Therefore the downtime due to page fault reduces effectively. In another work, Su et al. [11] proposed an algorithm to reduce the page fault during the post-copy migration by using the filter for page fault. This algorithm reduced the total migration time but the downtime was almost the same with the original post-copy. Chou et al. [12] proposed a hardware approach using high-performance Fabric-Attached Memory (FAM) to interconnect physical machines. Memory pages are dumped from source node to FAM during the migration, while the VM is running at destination node. It helps to release the source node and recover page faults from FAM faster. These works are trying to improve the critical page fault latency and total downtime.

Robustness has an essential role on migration performance, if either source or destination fails during the migration, the user may lose the whole service. Several works have tackled source node failure during post-copy migration. Dashpande et al. [13], [14] proposed a method to make post-copy more robust by scattering the memory pages from the VM to multiple nodes (consisting of the destination node and one or more intermediate nodes). Destination node can obtain the rest of the memory pages from the intermediate nodes. This mechanism helps memory pages to be evicted in shorter time from the source node which helps to lower migration failure in case of source node failure.

Meanwhile, Fernando et al. [15] focused on reducing the time of evicting the memory pages from source node with low impact on VM's performance during migration. In their algorithm, a snapshot of the VM's memory will be sent to a destination or a fail-over node before the migration. By this snapshot, recovering the VM after the failure is easier and faster. Later, Fernando et al. [16] investigated recovering the VM after failure. This approach uses reverse incremental check pointing called PostCopyFT. During the migration, the destination node sends the incremental changes and execution state to the source node periodically or upon external I/O events. It helps to recover the almost updated VM's memory after the failure of the destination node from the latest checkpoint.

III. System Model and Algorithm Design

In this section, we introduce the system model and the algorithm for robustly migrating a service within a slice at fog layer to maintain the *Quality of Service* (QoS) for IoT devices with low-latency sensitive services.

A. System Model

We consider a network slice at the fog layer which consists of several physical fog nodes that are distributed in a geographic area where users move around. The slice has a *Slice Manager* [17] (SM) which knows the positions of all the fog nodes, the current positions of all mobile users, and the fog nodes that are running the user services. The slice manager is responsible for deciding whether a service needs to be migrated, and selecting the destination and backup nodes.

The fog nodes can be homogeneous or heterogeneous, and each has an access point. The users are mobile IoT devices such as UAVs and *Autonomous Vehicles* (AV)s hosting their latency-sensitive services on fog nodes. We do not make any assumption about the mobility of the users, but we assume that the slice manager is able to obtain their trajectory at any

[2]Node failure may happen for different reasons such as flat battery or due to some hardware failure.

point in time. Fig. 2 illustrates the system model used in this paper.

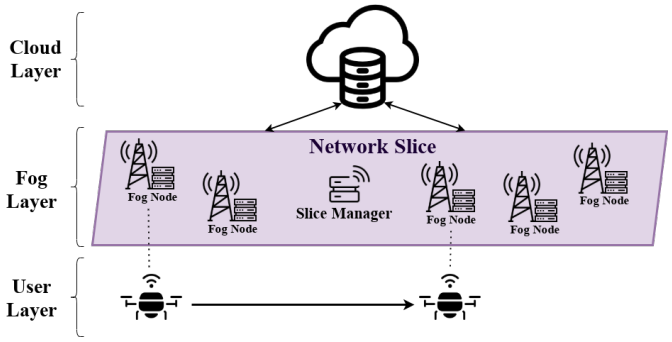

Fig. 2. System model: Mobile IoT devices hosting their services within a slice at the fog layer

When a user gets further from a fog node and there is a nearer fog node, its service should be migrated to the closer node to maintain low latency [18]. The migration process can be broken down into three steps: (i) deciding on when to commence migration, (ii) choosing the best destination node and (iii) performing the migration. In this work, we focus on the last step. Hence, we assume that the decision to start migration has already been made by the slice manager.

B. Preliminaries

Table I shows the notations that we use in this paper.

TABLE I
NOTATIONS

R	Bandwidth
S	Page size
S_{request}	Page fault request size
N	Total number of memory pages
$P_{\text{faultrate}}$	Page fault rate
$N_{\text{pagefault}}$	Total number of page faults
$N_{\text{prepaging}}$	Total number of pages that can be pre-paged
E	Processor state size
$T_{\text{pagefaults}}$	Total time of sending page faults
$T_{\text{prepaging}}$	Total time of actively sending memory pages
D	Downtime
T	Total Migration Time

1) Destination and Backup Node Selection: In the second step of the migration process, the SM ranks all destination candidate nodes to choose the best candidate as the destination for migration. We assume that the network slice has sufficient nodes such that there are at least two suitable candidate nodes for migration in the trajectory of the user. The candidates can be ranked based on QoS or QoE metrics such as bandwidth, latency between user and node, available resources, load and location. **The highest ranked candidate will be designated as the destination node and the second highest ranked candidate will be considered the backup node.** Our reason for choosing the second highest ranked candidate as the backup node is that if the destination node has indeed failed, then the new destination node (which is the backup node) will be the best option among all the remaining candidates to host the service. It is worth pointing out however that this new destination node may not provide the same level of QoS as the original destination node, but this is still better than losing the service for the user.

2) Pre-Paging: One downside of post-copy is the potentially high page fault rate. When a service encounters a page fault, its execution is temporarily suspended while the page is being fetched from the source node to the destination node. Page faults can cause latency fluctuations which can impact latency-sensitive services. Note that if N memory pages need to be migrated, then post-copy will incur $N_{\text{pagefault}}$.

To reduce latency fluctuations, we need to reduce page faults. To do this, our algorithm uses pre-paging similar to the technique proposed by Hines et al. [6] which can lower page fault rate by 79% compared to post-copy without pre-paging. With pre-paging, the total number of page faults is given by

$$N_{\text{pagefault}} = NP_{\text{faultrate}}.$$

As mentioned, every page fault will cause the service to pause. The amount of time that the service is paused is the sum of the time needed to send the page fault request to the source and the time needed to send the page to the destination. Note that this formulation ignores the time needed by the source to locate the requested page and other processing overheads. Let $T_{\text{pagefaults}}$ denote the total time due to page faults. Then we have

$$T_{\text{pagefaults}} = N_{\text{pagefault}} \left(\frac{S + S_{\text{request}}}{R} \right).$$

The downtime D is simply the time needed to send the execution state to the destination and the total time due to page faults:

$$D = \frac{E}{R} + N_{\text{pagefault}} \left(\frac{S + S_{\text{request}}}{R} \right). \tag{1}$$

Note that when pre-paging is employed, a fraction of the total number of page faults is pre-paged:

$$N_{\text{prepaging}} = N - N_{\text{pagefault}}.$$

The time needed to send one memory page from one node to another is simply S/R. From this, we can obtain the total time needed to perform pre-paging, denoted by $T_{\text{prepaging}}$, as

$$T_{\text{prepaging}} = N_{\text{prepaging}} \left(\frac{S}{R} \right).$$

We can now obtain the total migration time with pre-paging which is just the sum of total downtime and the time needed to perform pre-paging. This is given by

$$T = \frac{E}{R} + N_{\text{pagefault}} \left(\frac{S + S_{\text{request}}}{R} \right) + N_{\text{prepaging}} \left(\frac{S}{R} \right).$$

Since $N_{\text{pagefault}} + N_{\text{prepaging}} = N$, we can simplify the above as

$$T = \frac{E}{R} + N_{\text{pagefault}} \left(\frac{S_{\text{request}}}{R} \right) + N \left(\frac{S}{R} \right). \quad (2)$$

3) Fault Detection: During the migration process, the destination node will periodically send a heartbeat message to every other node involved in the migration. When no heartbeat is received, this indicates that the destination node has encountered some failure.

C. Robust Post-Copy Migration Algorithm

After selecting the destination and backup nodes, the migration begins. As shown in Fig. 3, in the first stage of migration (A), the source node suspends the container[3] and sends the execution state to both destination and backup nodes simultaneously. Both receivers send back an acknowledgement message to the source node indicating that they have received the execution state. From here on, a receiver sends back an acknowledgement message to the sender every time it receives a packet. With this mechanism, the sender will be aware that the receiver has received the packet.

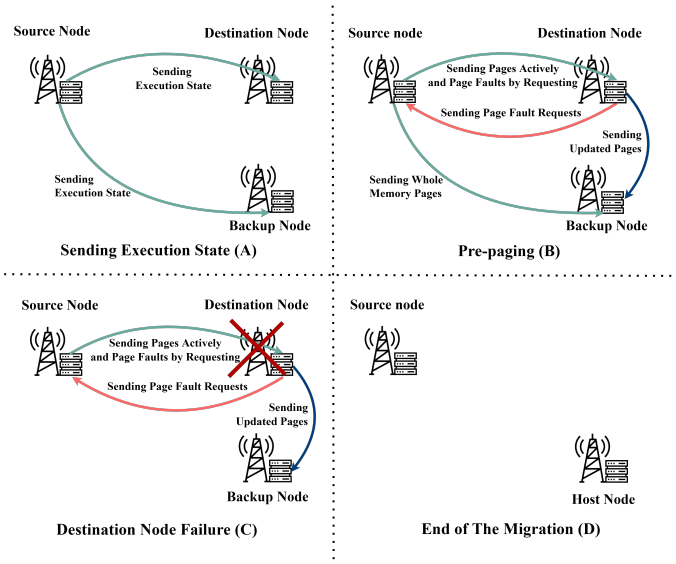

Fig. 3. Migration Scenario Where Destination Node Failed After Backup Node Have Already Received All Memory Pages.

In the second stage (B), the container at the destination node starts running and the user session is handed over to the destination node. The source node then starts to send all of the memory pages to the backup node. This part is similar to cold migration, as the service is paused in both source and backup nodes and the backup node receives memory pages sequentially. Every time the destination node encounters a page fault, it sends a page fault request to the source node. In response, the source node sends the requested page immediately, then actively pushes other pages close to the requested page following the pre-paging technique [6].

One important but subtle characteristic of this approach is that (assuming the bandwidths between the source to destination and source to backup are the same) the time required to send all memory pages to the backup node will be less than but close to the total migration time between the source and destination. This surprising behaviour is due to the fact that the transfer of the memory pages to the backup node is continuous, whereas the transfer of memory pages to the destination node is triggered by the occurrence of page faults.

To make the memory pages between the destination and backup nodes consistent, the destination node sends updated memory pages to the backup node. The memory pages from the destination will overwrite any previously received pages from either the source or destination nodes. The destination node also periodically sends the processor state's latest version to the backup node. If the source node is able to send all memory pages to the destination node, then the migration is complete and the source and backup nodes should terminate their respective containers.

The above discussion tackled the case where the destination did not fail during the migration. In the following, we will describe the algorithm used when the destination node fails. For convenience, we have separated the discussion into two cases: (i) when failure occurs after the backup node has received all memory pages; and (ii) when failure occurs before the backup node has received all memory pages.

1) Failure After Backup Node Received All Memory Pages: As can be seen in Fig. 3, in the third stage (C), the backup node has already received all memory pages from the source node as well as the updated memory pages from the destination node. Now, let us assume that the destination node failure occurs in this stage. After the failure occurrs, the backup node will resume the container and start working. The user session will be handed over to the backup node as the new host. Because the container at the new destination has all the memory pages and most of these pages and execution state are up-to-date, the user can run its service with less interruption. At this phase, the migration is effectively complete and the user continues running its service at the new host (D).

2) Failure Before Backup Received All Memory Pages: We now discuss the second case when failure occurs before the backup node receives all the memory pages from the source node. As can be seen in Fig. 4, in stage (C) destination failure has occurred. The failure may occur at any time from the start of the migration close to the end of the migration. When the failure occurs, the backup node will be assigned as the new destination. First, the user session is handed over to it and the container resumes. The main difference between this case and the first case is that the backup node does not have all the memory pages. Therefore, there is a need for the source node to send the remaining memory pages to the backup node which is now the new destination. The sending of the remaining memory pages will be done using pre-paging migration [6]. As shown in Fig. 4, in the last stage (D), post-copy migration with

[3]In the algorithm description, we consider container migration but it must be noted that the same algorithm can be applied to virtual machine migration as well.

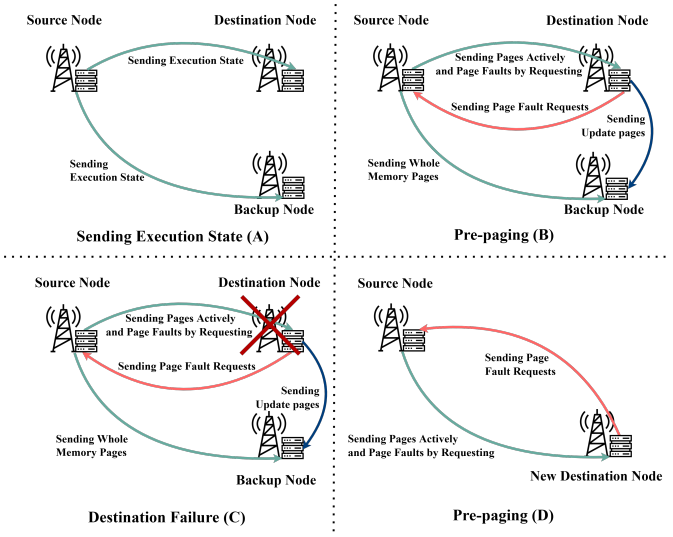

Fig. 4. Migration Scenario Where Destination Node Failed Before Backup Node Received All Memory Pages.

pre-paging is performed between the source node and new destination node. After the last memory page has been received by the latter, the former sends an acknowledgement message to the new destination about the completion of sending memory pages. Subsequently, the container at the source node will be terminated and migration will end successfully.

3) Improving Robustness Further: It is possible to improve the robustness of this algorithm by assigning a new backup node whenever the designated destination node fails. For this purpose, after the backup node is assigned as the new destination node, the third ranked candidate node can be assigned as the new backup to perform the robust algorithm as described above. To achieve efficient performance there is a need to consider the network conditions and also whether there are other nodes in the candidate list that are suitable to become the new backup.

IV. Simulation and Results

In this section, we present simulation results to validate the effectiveness of our proposed algorithm in the presence of failure at different points in the migration process.

A. Simulation Scenario

We considered a network slice at the fog layer consisting of homogeneous nodes. Each node can contain several containers. Each container hosts a service that is being utilized by a user. The bandwidth between the fog nodes is 50 MB/s (this is based on 5G network), and the container size is 512 MB. Furthermore, we assumed that destination candidate nodes were already ranked based on geographical distance and latency between the user and the candidate node, and bandwidth between the source and the candidate nodes. The highest ranked destination candidate has the lowest latency to the user, and it is in the user trajectory. For our simulations, we also considered different bandwidths between source to destination and source to backup. This is because the highest ranked candidate node may have the fastest bandwidth to the source compared to the other nodes.

B. Algorithm Simulation Model

We implemented a simulation model of the proposed algorithm in iFogSim2 Simulator [19], along with a non-robust migration algorithm for performance comparison purposes. Non-robust migration is essentially post-copy via pre-paging migration while robust migration is the proposed algorithm discussed in the previous section. The migration will transfer a container from a source node to a destination node without any backup node. When the destination node fails during the migration, the source node recovers the service to the state prior to migration. Hence, when the user reconnects to the source node, as mentioned above, the memory pages and the execution state are stale. After recovery, the source node starts another migration to the second highest ranked destination candidate node (new destination).

C. Simulation Parameters

We wanted to understand the performance of the proposed algorithm in the presence of failure so we varied the time at which the destination node failed during the migration process. In particular, we measured the failure time as a function of the fraction of memory pages that had been received by destination node during the migration. Failure may occur when between 10%–90% of the total memory pages have been received by destination node. We also varied the bandwidth between the source and backup node from 10 MB/s to 50MB/s. We run each scenario 30 times using the 9 different failure times and 5 different bandwidths, measuring the *total migration time* and *downtime*.

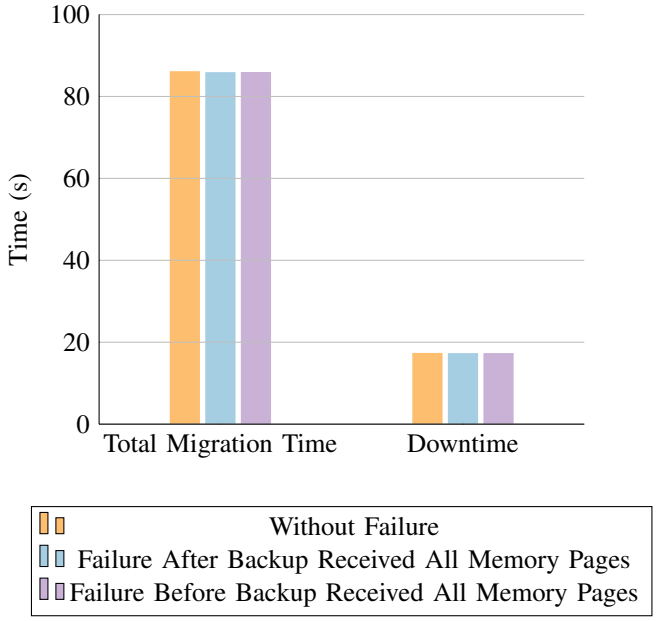

Fig. 5. Average of Total Migration Time and Downtime for Robust Migration

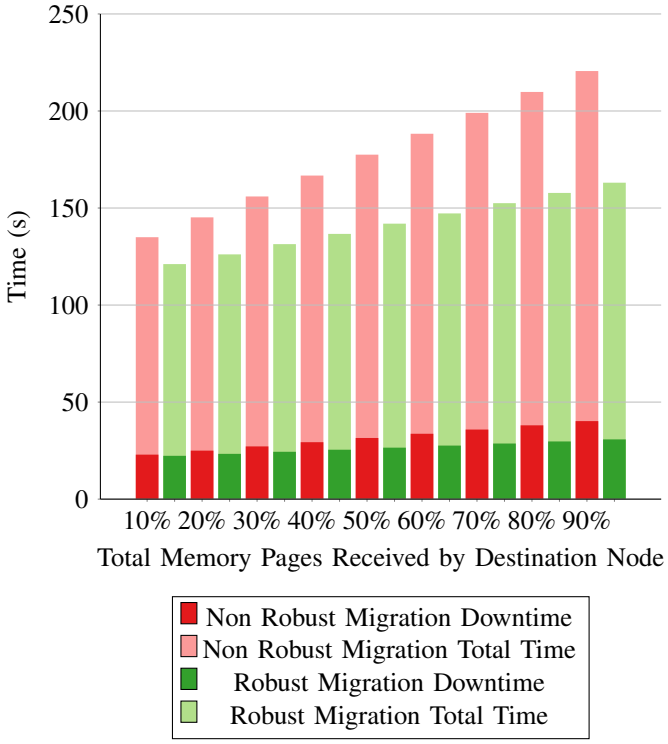

Fig. 6. Average of Total Migration Time and Downtime

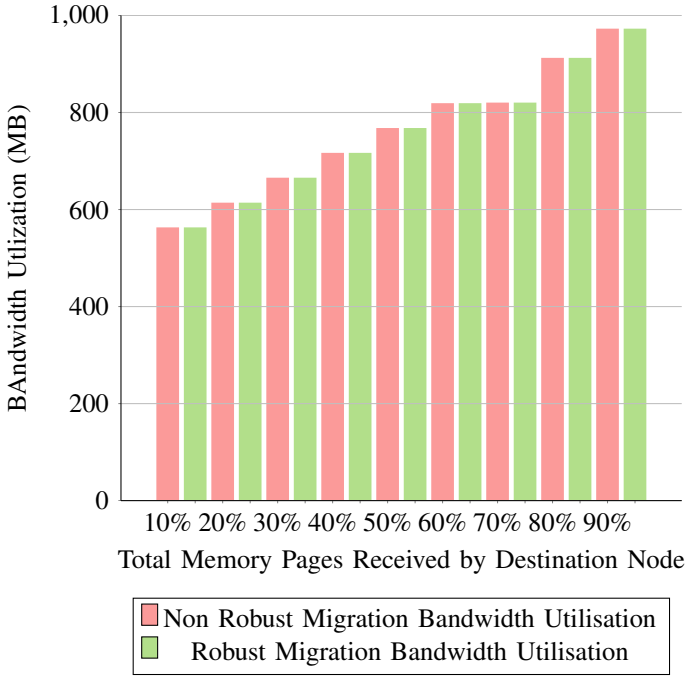

Fig. 7. Bandwidth Utilisation

Total migration time is the time from the start of migration until its completion. Downtime is the total time the service is paused due to handover and page fetching.

D. Robust Migration Performance

We investigated the performance of our scheme under different situations. We considered three different scenarios: (i) migration without failure; (ii) migration with failure after the backup node received all memory pages and (iii) migration with failure before the backup node received all memory pages. Bandwidth between the nodes are similar at 50 MB/s, and the container size is 512 MB. We ran each scenario 30 times and we computed the average of total migration times and downtimes with and without destination failure. The results are shown in Fig. 5.

It is noticeable that when the destination node fails, the total migration time and downtime with failure is close to the total migration time and downtime when there is no failure. This indicates that the robust feature of the algorithm is not adversely affecting its performance. These minimal differences are due to the fact that copying of memory pages to the backup node occurs simultaneously with the migration to the destination node. We anticipate that if the bandwidth between the source to destination and source to backup is shared, then the migration will take longer when employing the backup node.

E. Comparison with Non-Robust Migration

We computed the average of total migration time and downtime for non-robust migration in the presence of destination failure. Fig. 6 shows that robust migration had lower total migration time and downtime compared to non-robust migration. More significantly, the migration time and downtime of robust migration increased at a much lower rate compared to non-robust migration. When failure happens close to the start of migration (i.e., 10%), robust migration has 10% lower migration time and 2% lower downtime. When failure happens close to the end of migration (i.e., 90%), robust migration has 26% lower migration time and 23% lower downtime. The better performance of robust migration is due to the simultaneous transfer of memory pages from the source to the backup node, and updated memory pages from the destination to backup node.

To compare the cost incurred by robust migration and non-robust migration, we calculated the bandwidth utilisation by the two algorithms. As shown in Fig. 7, the bandwidth utilisation for both schemes are the same when there are failures. This result is surprising, as we expected robust migration to have higher utilisation due to the simultaneous transfer of memory pages from source to destination node and source to backup node. Note however that when failure occurs, non-robust migration has to perform recovery, i.e., transfer memory pages (from the start) to the new destination node. The utilisation of this recovery is exactly the same as the utilisation of the transfer of memory pages from source to backup node. Of course, when no failure occurs, the utilisation of robust migration is at most twice that of non-robust migration.

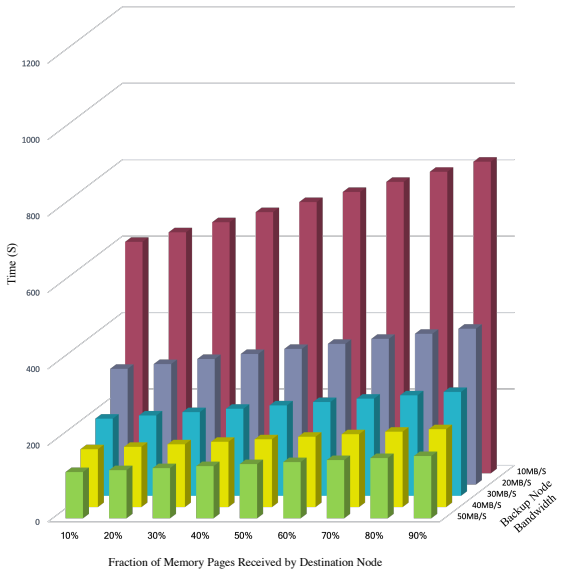

Fig. 8. Average of Total Migration Time for Robust Migration with Different Bandwidth

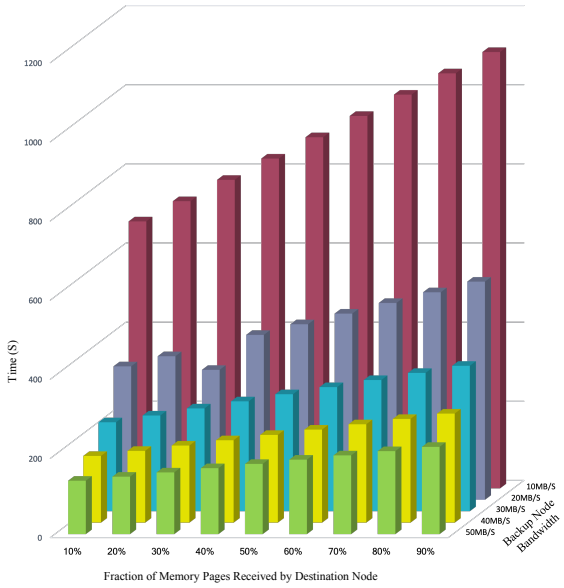

Fig. 9. Average of Total Migration Time for Non-Robust Migration with Different Bandwidth

F. Comparison with Non-Robust Migration with Different Bandwidths

We also computed the same performance metrics in Figs. 8, 9, 10 and 11 with different bandwidths between the source to destination node and source to backup node. We can see that in all scenarios, robust migration showed significantly better performance than non-robust migration. Moreover, when the difference between source to destination and source to backup bandwidth is highest (i.e. the latter is at 10 MB/s), the performance difference is also at the highest. This is a strong argument for the simultaneous transfer of memory pages between source to destination and source to backup. When the bandwidth of the latter is much smaller, it is better to start the transfer as early as possible such that when the destination node fails, the backup node would have received a higher number of the memory pages.

The stable performance of robust migration is not surprising as the backup node allows the migration process to continue after the failure. The performance of non-robust migration is adversely affected by the failure, and the effect is worse when the failure occurs later in the migration. This is because when the failure occurs at a later stage, the time spent on migrating is wasted as the source has to start all over again. The use of a backup node prevents this, although this comes at an additional overhead.

V. Conclusion

In this paper, we proposed a robust post-copy migration algorithm that is suitable for migrating latency-sensitive services in the fog layer. To the best of our knowledge, this algorithm is the first work that attempts to address the deficiencies of post-copy with regards to robustness within the network slice. Results show that the proposed algorithm does not adversely affect downtime and total migration time. Moreover, the proposed algorithm allows services to seamlessly continue in the event of destination failure. Compared to non-robust migration, robust migration reduced total migration time between 10-26% and downtime between 2-23%. Furthermore, when the bandwidth to the backup node is lower, robust migration provides further improvement in both metrics. Though the proposed algorithm may utilise more resources compared to non-robust post-copy, it is necessary to employ such a robust algorithm for scenarios where latency and robustness are crucial. In the future, we will conduct thorough evaluation of the algorithm using scenarios wherein node and network resources (e.g. CPU and memory) are heterogeneous.

References

[1] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[2] R. A. Addad, T. Taleb, H. Flinck, M. Bagaa, and D. Dutra, "Network slice mobility in next generation mobile systems: Challenges and potential solutions," *IEEE Network*, vol. 34, no. 1, pp. 84–93, 2020.

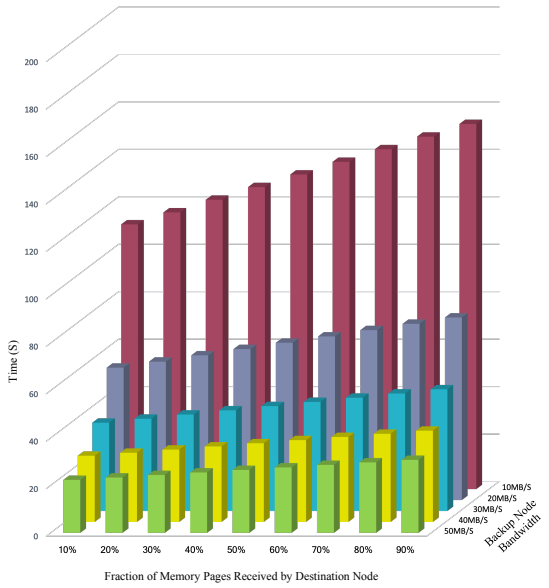

Fig. 10. Average of Downtime for Robust Migration

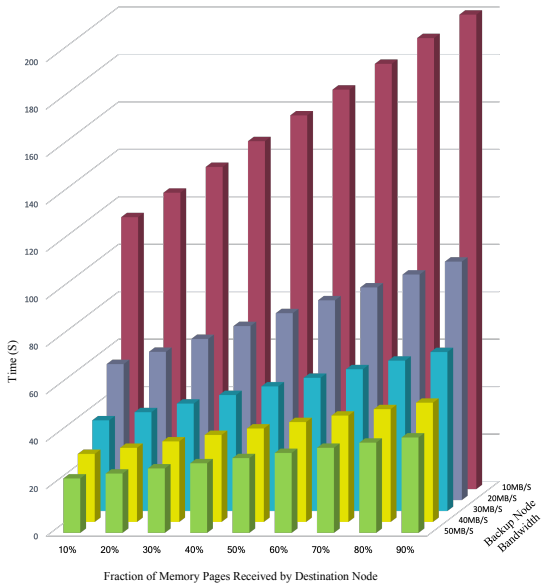

Fig. 11. Average of Downtime for Non-Robust Migration

[3] B. E. Mada, M. Bagaa, T. Tale, and H. Flinck, "Latency-aware service placement and live migrations in 5g and beyond mobile systems," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pp. 1–6, 2020.

[4] R. Bruschi, F. Davoli, P. Lago, and J. F. Pajo, "Move with me: Scalably keeping virtual objects close to users on the move," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2018.

[5] S. Filiposka, A. Mishev, and K. Gilly, "Community-based allocation and migration strategies for fog computing," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, 2018.

[6] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, p. 14–26, July 2009.

[7] F. Zhang, G. Liu, X. Fu, and R. Yahyapour, "A survey on virtual machine migration: Challenges, techniques, and open issues," *IEEE Communications Surveys Tutorials*, vol. 20, no. 2, pp. 1206–1243, 2018.

[8] T. Le, "A survey of live virtual machine migration techniques," *Computer Science Review*, vol. 38, p. 100304, 2020.

[9] D. Fernando, J. Terner, K. Gopalan, and P. Yang, "Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pp. 343–351, 2019.

[10] Z. Shan, J. Qiao, and S. Lin, "Fix page fault in post-copy live migration with remotepf page table assistant," in *2018 17th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pp. 40–43, 2018.

[11] K. Su, W. Chen, G. Li, and Z. Wang, "Rpff: A remote page-fault filter for post-copy live migration," in *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pp. 938–943, 2015.

[12] C. C. Chou, Y. Chen, D. Milojicic, N. Reddy, and P. Gratz, "Optimizing post-copy live migration with system-level checkpoint using fabric-attached memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 16–24, IEEE, 2019.

[13] U. Deshpande, Y. You, D. Chan, N. Bila, and K. Gopalan, "Fast server deprovisioning through scatter-gather live migration of virtual machines," in *2014 IEEE 7th International Conference on Cloud Computing*, pp. 376–383, IEEE, 2014.

[14] U. Deshpande, D. Chan, S. Chan, K. Gopalan, and N. Bila, "Scatter-gather live migration of virtual machines," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 196–208, 2015.

[15] D. Fernando, H. Bagdi, Y. Hu, P. Yang, K. Gopalan, C. Kamhoua, and K. Kwiat, "Quick eviction of virtual machines through proactive snapshots," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 156–157, IEEE, 2016.

[16] D. Fernando, J. Terner, K. Gopalan, and P. Yang, "Live migration ate my vm: Recovering a virtual machine after failure of post-copy live migration," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 343–351, IEEE, 2019.

[17] A. A. Barakabitze, A. Ahmad, R. Mijumbi, and A. Hines, "5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106984, 2020.

[18] H. Elazhary, "Internet of things (iot), mobile cloud, cloudlet, mobile iot, iot cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions," *Journal of Network and Computer Applications*, vol. 128, pp. 105–140, 2019.

[19] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *Journal of Systems and Software*, vol. 190, p. 111351, 2022.