MS2M: A message-based approach for live stateful microservices migration

Hai Dinh-Tuan
Service-Centric Networking
Technische Universität Berlin
Berlin, Germany
hai.dinhtuan@tu-berlin.de

Felix Beierle
Institute of Clinical Epidemiology and Biometry
University of Würzburg
Würzburg, Germany
felix.beierle@uni-wuerzburg.de

***Abstract*—In the last few years, the proliferation of edge and cloud computing infrastructures as well as the increasing number of mobile devices has facilitated the emergence of many novel applications. However, that increase of complexities also creates novel challenges for service providers, for example, the efficient management of interdependent services during runtime. One strategy is to reallocate services dynamically by migrating them to suitable servers. However, not every microservice can be deployed as stateless instances, which leads to suboptimal performance of live migration techniques. In this work, we propose a novel live migration scheme focusing on stateful microservices in edge/cloud environments by utilizing the underlying messaging infrastructure to reconstruct the service's state. Not only can this approach be applied in various microservice deployment scenarios, experimental evaluation results also show a reduction of 19.92% downtime compared to the stop-and-copy migration method.**

***Index Terms*—micorservices, live migration, service orchestration, asynchronous messaging**

I. Introduction and Motivation

In the last few years, microservice architectures have attracted attention from both academia and industry as a novel approach to software development. In fact, large cloud service providers such as Amazon or Netflix have widely adopted this new architecture to serve millions of users on a daily basis [1]. In essence, this software design encourages the decomposition of monoliths into smaller, independently deployable units, each provides one or several services [2].

In the *Twelve-Factor App methodology* [3], the authors have described an ideal scenario, in which all microservices should be designed and implemented as stateless instances. Since stateless services do not have the responsibility to manage state data internally, and no prior knowledge is needed, each request can be processed by any service instance. This enables applications to achieve a very high level of scalability and resiliency, because multiple service instances can be quickly deployed to handle peak loads. Similarly, failed service instances can also be replaced by launching other replacement instances. However, it is not always feasible to design a system based entirely on stateless services [4]. Over the years, several workarounds have been proposed to delegate state data management to an external entity. For example, a service can adopt the *client side state management technique* to store state information in the clients instead. Another option is to constantly save and load service states from an external database. However, in both cases, additional overheads caused by state data exchanges reduce the application's latency performance [5] and generate more network loads.

In distributed systems such as microservices, a reliable communication channel among services plays a crucial role to orchestrate services, especially in the *event-driven* design, where all the events and state information are constantly exchanged among service instances. Therefore, this communication channel can be considered as the backbone of a microservice-based application, and in fact, some existing frameworks even require the developers to design the data communication before implementing the services [6]. Thus, in this paper, we investigate how to exploit the already mature communication infrastructure for microservices to solve the state management problem and develop an efficient migration scheme.

This paper focuses on service migration, where the problem of state management is most prominently presented. In recent years, with the popularity of cloud-based services as well as the advent of new concepts such as osmotic computing [7], multi-access edge computing [8], service migration has been increasingly discussed in the literature as a way to not only achieve resource management goals (power management, resource load balancing) but also to improve application's flexibility, fault tolerance, etc. This paper describes a novel approach to service migration using the existing message exchange mechanisms in many microservices-based applications to support stateful microservices migration without adding a dedicated state management entity. We name our proposed approach MS2M, which is an acronym for *Message-based stateful microservice migration*.

In this work, we implemented a prototype using a remake version of Flappy Bird – one of the most iconic mobile games originally launched in 2013. This interactive game is chosen so that we can easily demonstrate how the migration process affects the game quality. We have modified the codebase with new components to evaluate our migration scheme. While the user interacts with the browser-based component, one cloud-based component is migrated to a cloud region with better latency performance. The goal is to deliver a seamless experience for the players, i.e., the game is not interrupted

during the migration process.

The contribution of this work is therefore two-fold: (1) design and implementation of a novel technique to facilitate stateful microservices portability using the existing message exchange mechanism, and (2) a performance evaluation of a prototype performed in a typical cloud environment.

In the remainder of this paper, Section II discusses the existing migration techniques that have been found in the literature, followed by a detailed description of the proposed concept for service migration using asynchronous messaging in Section III. Section IV briefly explains the evaluation setup, and the results are also presented. Finally, we discuss the results and conclude this work in Section V.

II. Related work

A. Service migration techniques

Currently, microservices are usually encapsulated into virtual machines and containers to simplify the deployment and avoid interoperability issues [9]. Therefore, we focused on related migration techniques for those deployment technologies.

As virtual machines have been widely utilized for a long time, virtual machines migration is a well in-depth analyzed topic with plenty of proposals [10], [11]. Virtual machines persist their states in virtual disks, and they are migrated across platforms using various tools, both proprietary like Hyper-V Live Migration or third-party tools. The simplest approach is *cold migration* or *stop-copy*, in which the execution is first stopped at the source, after the state is transferred and restored at the destination, the execution continues at the new location [12]–[14]. As a result of its simplicity, this approach minimizes the amount of data transfer between source and destination, while maintaining a small total migration time in comparison to more sophisticated techniques. However, both downtime and total migration time are proportional to the amount of data that need to be transferred. In addition, there is a downtime period where no client request can be processed. To minimize this downtime period, there are several live migration schemes proposed such as pre-copy migration [15], post-copy migration [16], or hybrid [17], which are considered as the foundation for a number of techniques proposed recently. Each of those has a slightly different approach, which is summarized in Figure 1.

At the moment, containers have slowly replaced virtual machines to become the de facto standard to deploy cloud services. At the edges, where the resources are limited and heterogeneous, it is even more favorable to deploy containerized services due to its low virtualization overhead and fast responsiveness. In certain cases, containers can even achieve near-native performance [18], [19]. However, containers migration in general and containerized microservices migration in particular have not been well mentioned in published works so far [20]. Over the years, several new techniques have been developed and in the following, we mention several state-of-the-art techniques developed specifically for container live migration, namely Virtuozzo, Picocenter, ClusterHQ's Flocker, and Voyager.

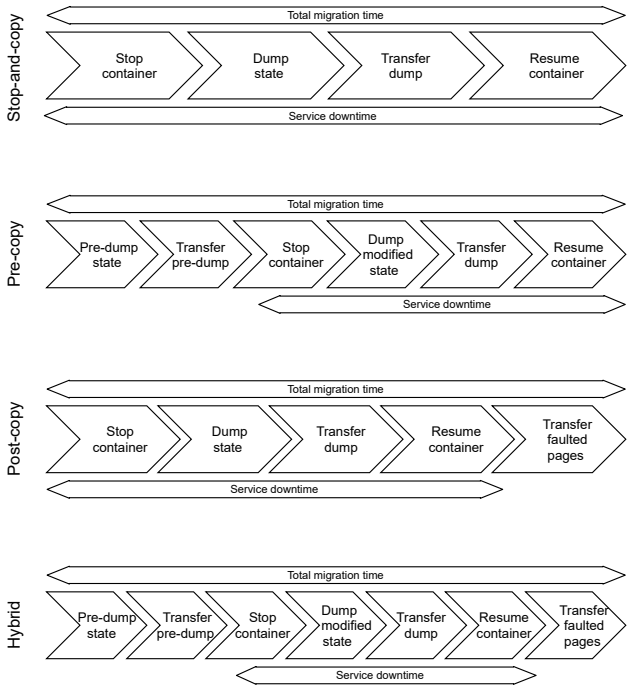

Fig. 1. Overview of stop-and-copy, pre-copy, post-copy and hybrid migration techniques.

Virtuozzo [21] provides a bare-metal virtualization solution that supports containers and can facilitate live migration. Firstly, the container's file system and virtual memory are transferred to the target host. After this stage, Virtuozzo freezes all container processes and disable networking. As the container is stopped, its memory state is dumped into a file, and this file is transferred to the target host. From this moment, one or several synchronization rounds take place, in which changed memory page and disk block since the last transfer are migrated. When this iterative process ends, the container is resumed at the target host. The implemented technique is based on the post-copy technique mentioned above, which can achieve a relatively short downtime and requires a small amount of data to be transferred. However, like any other post-copy migration technique, it is only feasible when the amount of changed memory pages and disk blocks (*deltas*) is small, thus outage time imperceptible. Therefore, if there is a high latency connection between source and target host, as well as when the page dirtying rate is high, for example in data intensive applications, the total migration time can be too long to be acceptable.

Picocenter [22] has been built with the ability to swap out inactive containers from the cloud to object store (Amazon S3) and swap in them back on demand. Under the hood, it uses Checkpoint/Restore In Userspace (CRIU) to capture memory-state and *btrfs file system* to store the persistent state. In addition, *ActiveSet* was proposed to allow memory pages

to be on-access and lazily restored.

Flocker by ClusterHQ is an open-source container data volume manager developed specifically for Docker containers. It supports migration for network-attached storage backends like Amazon EBS, Openstack Cinder, VMware vSphere etc., by re-attaching these network storage for containers. However, locally attached volume migration is supported only for ZFS filesystem.

Voyager [23] is a file system– and vendor–agnostic migration service developed for OCI (Open Container Initiative)–compliant containers. Voyager enables just-in-time live container migration with short downtime, by discovering all data endpoints of a container and migrating in-memory state via CRIU. Network attached storage is manually unmounted from the source host and mounted again at the target host. With a union view of the data between the source and target hosts, Voyager containers can resume an operation instantly on the target host, while performing disk state transfer either on-demand (Copy-on-Write) or through lazy replication.

B. Asynchronous messaging in microservices-based applications

Considered as a new architectural style for cloud applications, microservices are designed to be self-contained, so that each service can be developed and deployed independently. To extend this independence into operation, the microservices usually rely on an external communication infrastructure for exchanging data among services. This design allows developers to focus on the actual microservices' business logic rather than on implementing mechanisms for sending and receiving data. Among others, asynchronous messaging is widely implemented for microservices, as this communication scheme can ensure that no microservice has to wait for another service's response [24]. In this design pattern, all communication among services are encapsulated in messages, and they are handled asynchronously by a message broker, hence the name "asynchronous messaging". This concept is also known as "smart endpoints and dumb pipes concept" [25], in which the term "dumb pipes" refers to simple and lightweight asynchronous messaging like Advanced Message Queuing Protocol (AMQP) or Message Queuing Telemetry Transport (MQTT), both are widely supported by a number message brokers such as RabbitMQ or ZeroMQ [26].

In addition, to decouple client apps from microservices, minimize the service calls and reduce the attack surface, microservices are usually not built with public endpoints, i.e., external clients cannot consume services directly, but rather through an API gateway [27]. Therefore, a major proportion of communications among microservices are done internally, especially in event-driven design, where all events are published and exchanged across microservices as messages. Those messages are exchanged via publish/subscribe queues provided by the message broker. This means, in such a design, the message broker plays a crucial role in coordinating the entire cluster of microservices. Given that asynchronous messaging in general and the message broker in particular both are widely used in various microservice implementations [28], this work's main idea is to make use of those components to reproduce the states of migrated services during the migration process.

III. Concept and Design

Compared to the above-mentioned techniques, this work proposes a novel migration scheme named MS2M, which also utilizes CRIU to create checkpoints and restore the containerized microservice at the target host. However, the novelty of this work lies in the *state synchronization mechanism*. Instead of transferring service states directly by copying in-memory data, which is proved complex by previous works, this proposal indirectly rebuilds the state of the migrated instance by replaying incoming messages during the migration period. In this section, we present in detail how this idea works and discuss several of its fundamental characteristics. An in-depth discussion based on experimental evaluation results is given in the next section.

In this proposal, we differentiate three main involving actors, which are described as below:

- **Source host:** The server, where the microservice is originally hosted.
- **Target host:** The new server, where the microservice will be migrated to.
- **Migration manager:** This entity's main responsibility is to monitor and coordinate the migration process among source host, target host and the microservice. This can be deployed as another microservice within the application itself, thus enabling the self-management capability of the application.

The three actors interact with each other during the migration process, which can be divided into five main phases: Service checkpoint creation, Checkpoint transfer, Service restoration, Messages replay, and Finalization. These interactions are illustrated in Figure 2 while the details of each phase are explained below:

1) **Checkpoint creation:** The Migration Manager sends a *ServicePauseRequest* to the Source Host to temporarily pause the service instance and prepare to create a service checkpoint. During this preparation phase, the service is unsubscribed from the *main message queue* before a checkpoint is created. This is necessary to prevent the newly restored service at the Target Host from consuming the same message queue immediately after restoration. After the checkpoint is created, a *secondary message queue* is generated, which keeps a copy of every message coming to the *main message queue* since the migration initiation. Then, the service instance at the Source Host is resumed and continues to operate normally. Upon this phase completion, the Migration Manager is informed to proceed to the next phases.
2) **Checkpoint transfer:** During this phase, the checkpoint is transferred to the Target Host. This process happens in the background without interrupting service operation. The incoming requests are handled by the original service instance as normal.

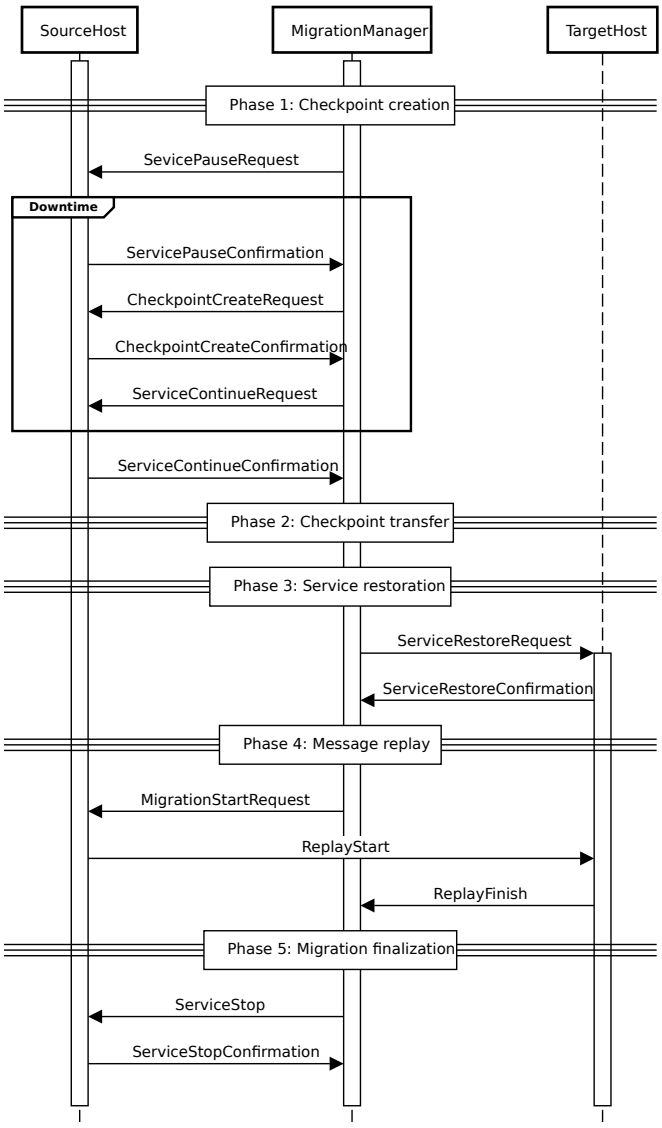

Fig. 2. Detailed migration process of MS2M.

3) **Service restoration:** After the checkpoint is fully transferred to the Target Host, the service instance is restored and configured to subscribe to the *secondary message queue*. When the newly restored service instance functions properly, the Migration Manager is informed by a confirmation from the Target Host.
4) **Message replay:** In this phase, the restored service instance at the Target Host starts processing messages in the *secondary message queue* without any output to synchronize the state with the original instance. Because all the message that arrives since the migration initiation are duplicated from the *main message queue* to the *secondary message queue*, the original service instance and the migrated service instance have the exact same input. During this message replaying phase, the original service instance at the Source Host still operates normally. By this design, the migrated service instance can synchronize the state with the original service instance while the incoming requests are handled normally by the original instance, therefore it helps to minimize the downtime. When this phase ends, the original instance notifies the migrated instance the *LastMessageProcessedID*, to indicate which messages have been processed successfully.
5) **Finalization:** After replaying the *LastMessageProcessedID*, the migrated instance switches back to the *main message queue*. In addition, from this point in time, the processed output from the migrated instance is sent normally to the message broker. The original instance stops and the migration process is finally completed.

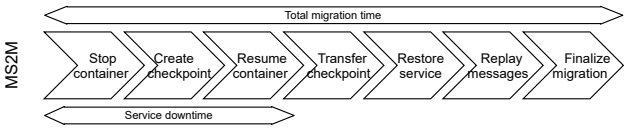

Fig. 3. Overview of the proposed MS2M scheme.

Figure 3 summarizes the above process, showing the Total migration time and the Service downtime of MS2M. Compared to the similarly illustrated techniques in Figure 1, one can immediately recognize the key difference: The dump transfer process in other techniques is included in the service downtime, while in MS2M, the checkpoint transfer phase is not, which can potentially lead to significant shorter downtime. This is made possible because in MS2M, the state difference is tracked indirectly by capturing messages to the *secondary message queue*. Therefore, the original service instance can be resumed immediately after checkpoint creation, and the checkpoint can be transferred to the Target Host in the background. The state difference occurs during the period between checkpointing and service instance restoration at the Target Host can be easily re-synchronized by replaying messages from the *secondary message queue*. During this process, the original instance still processes messages normally, and this task will only be handed over to the new instance when the message replay phase ends, i.e., when the service states in the original instance and the new instance are synchronized.

In addition to shorter service downtime, compared to previously discussed approaches, this proposed scheme has several additional advantages:

Firstly, this is a *communication–flow–aware approach* for service migration. Rather than focusing on transferring in-memory data, which is proved to be complex in previous works, MS2M employs the message broker to support the state synchronization process indirectly. By doing that, the application itself can take control over the decision when and where to migrate a particular microservice to maximize the performance. This means during the development phase, the developers based on the application's specifics can already decide what is the optimal operation conditions for the mi-

croservices.

Secondly, the downtime is minimized to the time required to prepare the service for checkpoint and the checkpoint process itself. It means the downtime does not depend on the network utilization, which is hard to predict during peak network traffic hours.

Lastly, in this concept, we took CRIU as an example for a container checkpoint/restore engine. However, there are several other tools that provide similar functionalities for other deployment strategies. Depending on the particular scenario, software architects can employ the most appropriate tool. The heart of this migration scheme is the coordination among microservices hosts and the replaying phase, which are both performed by the Migration Manager. This means, the exact approach can be applied for microservices–as–containers, microservices–as–virtual machines and other deployment options.

However, this proposal also comes with certain drawbacks. One of the major ones is the unpredictable nature of the replaying process. In case the incoming message rate is higher than the processing rate of the microservices during the replay phase, this iteration could theoretically never end. This is not desirable because during this phase, at least double the resources are occupied for both service instances to process the same request twice. On top of that, both the Migration Manager and the additional queue create overhead in the communication channel. Another possible issue is that this migration scheme doesn't guarantee the completion of migration if the original instance stops working during the process. Last but not least, there might be a degraded service period after the service instance finishes replaying messages, when the message processing latency is higher than normal.

In this section, we have presented a detailed description of the proposed migration process together with some preliminary analysis on its advantages and disadvantages. In the next sections, with the help of an interactive game prototype, we provide a more thorough understanding of the proposed migration techniques across several criteria such as latency performance.

IV. Experimental evaluation

A. Experimental Setup

For demonstration and evaluation purposes, we implement an interactive game prototype. This is a remake version of an iconic mobile game named Flappy Bird. In this arcade-style game, there is a bird that moves persistently to the right and automatically descends. The player should constantly enter inputs via the keyboard to keep the bird flying and avoid the pipes as shown in Figure 4. Whenever the player creates an input using the space bar, the bird ascends. The score is calculated based on the number of pipes that the bird can safely pass.

To evaluate our MS2M migration technique, a new cloud-based microservice has been added to persist state data from the game session, including players' game settings and scores. While the user interacts with the browser-based component,

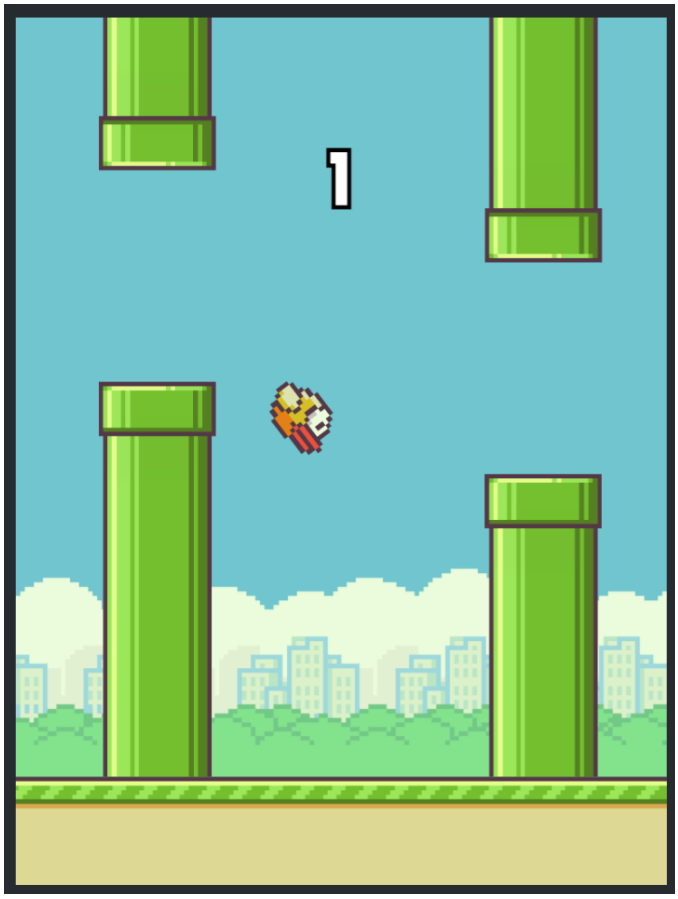

Fig. 4. The player's mission is to keep the bird flying without hitting the pipes.

the microservice is migrated to the cloud region with better latency performance and related data is recorded via various tracers. The detailed evaluation of these data is presented later in the second part of this section.

The whole application are deployed as four virtual machines on Google Cloud Platform (GCP) with the following configurations:

1) **Broker:** configured using GCP's e2-medium (2 vCPUs, 4 GB memory).
2) **Web App:** configured using GCP's e2-micro (2 vCPUs, 1 GB memory).
3) **Source and Target host:** Two virtual machines located in Belgium and the USA to host the containerized microservice. Both are configured using e2-medium (2 vCPUs, 4 GB memory)

All virtual machines are deployed with Ubuntu 20.10 Groovy. The setup uses Podman 3.1.0 as container engine and RabbitMQ 3.8.11 as message broker. The Migration Manager is deployed as a microservice, and it communicates with the host via Podman's API. This API supports executing CRIU's command directly, such as *create checkpoint*, *start container*, *stop container*, *resume container*. Regarding the virtual machines' locations, only one of them (Source Host)

is located in the USA, while all other machines are hosted in Belgium.

The evaluation test is performed by having a person playing the game on a computer. During this game session, a script is run to automatically migrate the microservices back and forth between GCP's computing region in the US and Belgium. We ran in total 50 migration tests, but because the network topology and load on GCP cluster might affect the evaluation results, we only consider the migrations in one direction from the USA to Belgium (in total 25 data points).

B. Evaluation results

As in any migration technique, the delay performance is one of the most crucial evaluation criteria. Similar to many other previous works when evaluating a certain technique, we evaluate two important values, which are the *Total migration time* and *Service downtime*.

The *total migration time* is counted from the initiation of the migration until the service is fully migrated and works properly at the new location, while the *service downtime* corresponds to the duration in which the service is completely unavailable. Long service downtime is obviously not favorable, as more user requests will be rejected in this case. From users' perspectives, the total migration time is less significant, however, a long total migration time results in more resource utilization and a longer service degradation period. This means, in the scenario where multiple simultaneous migration processes take place, a long migration time can lead to excessive resource usage. Therefore, an ideal migration technique should be able to minimize both of those values.

Based on those definitions, in this particular example, these periods are defined as follows:

1) The *total migration time* starts when the Migration Manager initiates the migration process by sending the *ServicePauseRequest* to the Source Host. This period ends with the stop of the original service instance at the Source Host while the service is resumed by the new service instance at the Target Host.
2) The *service downtime* in this scenario equals to the Service checkpoint creation time, in which the original service instance is offline and there is no service instance serving the incoming request. However, it is important to note that due to the asynchronous nature of the message broker, no user request is rejected, thus the user perceives this downtime as a degraded service period when it takes longer as usual to process requests.

To evaluate our proposed MS2M, in the prototype we also implemented the stop–and–copy migration technique as the baseline for comparison. The details of stop-and-copy migration technique implemented in this test are summarized in Figure 5 below.

For the evaluation, we collect data from 25 evaluation iterations, each iteration is done by both migration techniques implemented. The detailed results are presented in Figure 6 and Figure 7.

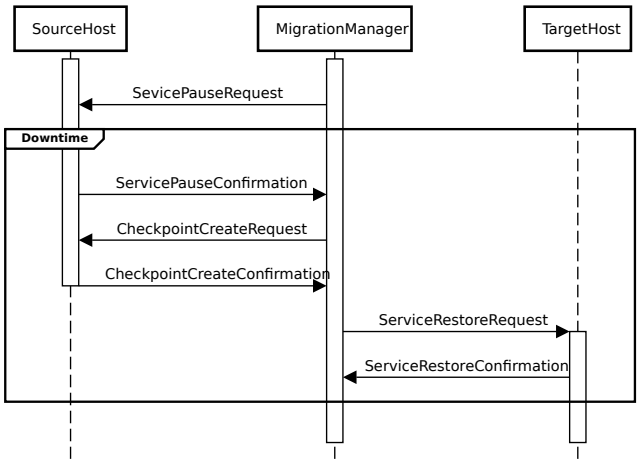

Fig. 5. Stop-and-copy process.

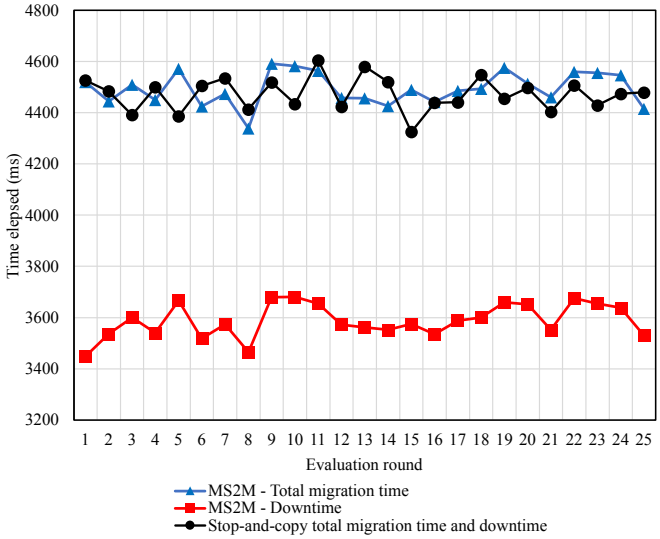

Fig. 6. Overall time performance comparison between stop-copy method and the proposed method.

The results show a similar total migration time in both schemes. While the stop-and-copy migration needs 4472.72ms, the proposed method takes slightly longer – 4852.86ms on average (±8.50% more) – to complete the migration process. Interestingly, the evaluation confirms our expectation of a shorter downtime by MS2M: The new scheme has only 3581.84ms downtime, yields a 19.92% reduction from the stop–and–copy scheme.

To gain a further understanding of the delay performance of the proposed technique, we also performed a more detailed time analysis to reveal the distribution of elapsed time in different phases as shown in Figure 8. Data for five main phases are collected, which are *Service pause* (time elapsed to pause the original instance), *Service checkpoint* (time elapsed to create the instance's checkpoint), *Service continuation* (time elapsed to continue the original instance after checkpoint), *Service migration* (time elapsed to migrate the checkpoint),

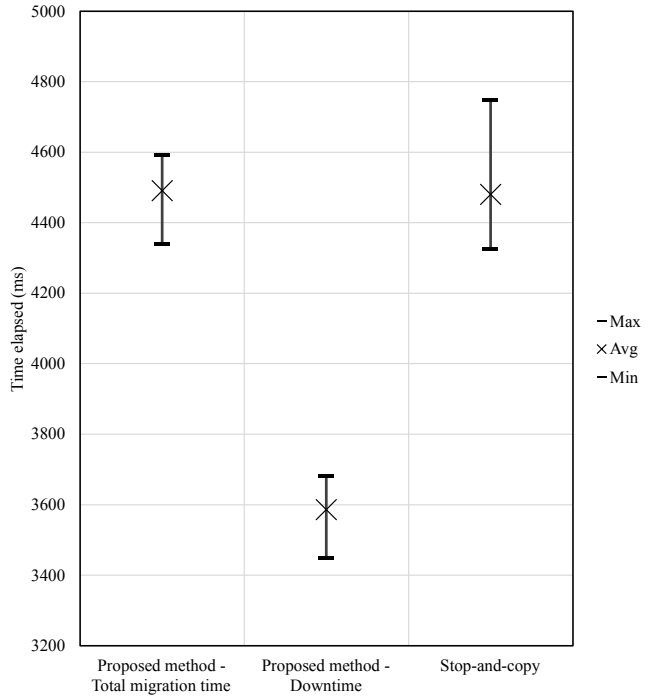

Fig. 7. Overall time performance comparison between stop-copy method and the proposed method.

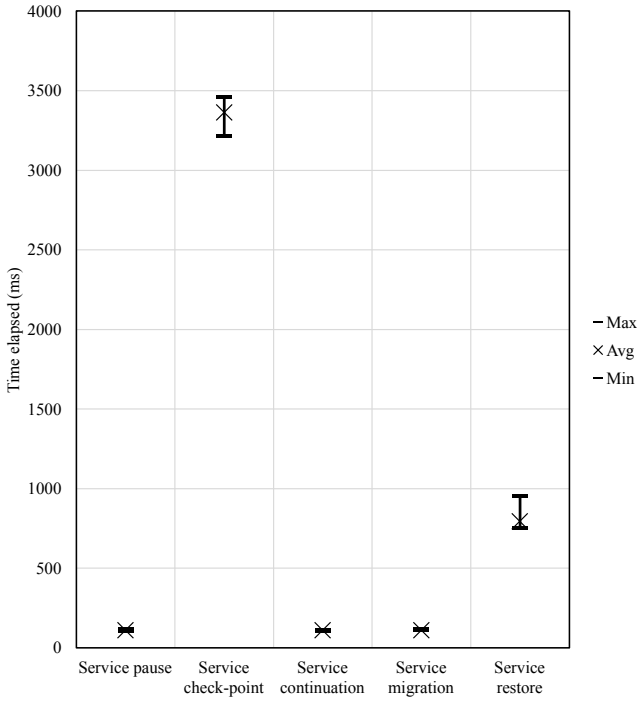

Fig. 8. Time performance analysis of the proposed MS2M method.

and *Service restoration* (time elapsed to restore the instance at Target host). While *Service pause*, *Service continuation*, and *Service migration* account for negligible proportions; i.e. 1.23%, 1.23%, and 9.54% respectively, the Service checkpoint and Service restoration process account for 63.04% and 24.97% of the total migration time.

These experimental results complement the advantages presented in Section III, demonstrating the feasibility of MS2M to use the message queues to support the migration process. By using an additional message queue, this setup can reduce the downtime by nearly 20%. This means that fewer client requests arrive without being processed. In addition, this newly proposed scheme can maintain a comparable total migration time with only 8.50% difference compared to the stop–and–copy method. These initial results can be used to prove the feasibility of our proposal in a cloud-based microservices environment.

However, there are also some important remarks. Firstly, although the experimental results show a comparable total migration time between the proposed scheme and the stop-and-copy method, it is necessary to note that in case the message arrival rate is high, the results might look entirely different. This is similar to the *high page dirty rate* scenario in the post-copy scheme: The higher this rate is, the longer it takes to finish the migration. Especially when the message arrival rate is higher than the message processing rate, it could theoretically take infinite time to finish the message replay phase, resulting in an unacceptable long total migration time. As mentioned before, this could become critical as more resources are utilized, hence the need to minimize this period.

In addition, the checkpoint and restoration process accounts for a major proportion of the migration time (88.04%). As pointed out by previous work, the checkpoint time increases linearly with the size of applications' memory state [23]. This suggests microservices with a larger memory footprint can also reduce the performance of this migration technique.

In this particular use case of an interactive game, both the amount of incoming data and its rate to the microservice are not significantly high. This explains why the total migration time of the two migration techniques recorded in this evaluation are not much different. Therefore, it should be noted that depending on the nature of the application, the total migration time could be longer compared to stop-and-copy scheme.

V. Conclusion

An ideal service-oriented microservices-based application should be developed as stateless to separate data management responsibilities from services, thus reducing resource consumption and improving the overall scalability. However, we have pointed out that this is not always feasible, i.e., stateful microservices are essential in certain applications. However, dealing with those stateful services requires special attention to those state data, which is reflected in the case of service migration. This paper presents the design, implementation, and evaluation of a novel approach for live stateful microservice migration. It shows how the asynchronous message broker can be utilized to support the migration process and enable service migration without the need to directly handle in-memory data. By using the message broker to spawn a new

message queue and indirectly track the state difference during migration, this method reduces the downtime by as much as 19.92% compared to the stop-and-copy method. Although the prototype used in this work is deployed as containers, the concept of using the message broker to support the restoration of state information can be generalized to apply to any event-driven microservices regardless of deployment strategy.

However, there are still some characteristics of the message replay process, which we need to pay special attention to. Firstly, if the incoming message rate is high, especially in the extreme case, when this value is higher than the processing rate, it could take a long time to finish the migration process, leading to excessive resource usage. In addition, services with a large memory footprint could also reduce the performance of this migration technique, because the service checkpoint and restore process largely depends on the service size, and they are both main contributors to the total migration time.

The results taken from this work lead to several possible future research directions to address open questions. An in-depth comparison with more recent migration techniques offered by cloud platforms such as Kubernetes would bring new insights, especially in real world scenarios. We also plan to further investigate possible solutions to optimize the checkpoint and restore processes, for example with an *incremental checkpoint* technique. In addition, the time performance of this proposed scheme needs more extensive analysis, especially the message replay process. Another use case with a high frequency of messages exchanged among various service types might help to accurately characterize this method's performance.

VI. Acknowledgement

We would like to show our gratitude to our students Xhorxhina Taraj, Spoorthi Satheesha, Tuan Anh Tran, Zubair Hossain, Sanjeet Raj Pandey, and Dimitrios Peppas for assisting us during the course of this research.

References

[1] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[2] M. Fowler. (2014) Microservices - a definition of this new architectural term. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] A. Wiggins, "The twelve-factor app, 2012," *Saatavissa (viitattu 26.3. 2016) http://12factor. net*, 2012.

[4] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency," *IEEE Software*, vol. 35, no. 3, pp. 63–72, 2017.

[5] J. Ingeno, *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.

[6] H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, "Development frameworks for microservice-based applications: Evaluation and comparison," in *Proceedings of the 2020 European Symposium on Software Engineering*, 2020, pp. 12–20.

[7] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 2016.

[8] H. Tanaka, M. Yoshida, K. Mori, and N. Takahashi, "Multi-access edge computing: A survey," *Journal of Information Processing*, vol. 26, pp. 87–97, 2018.

[9] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study." in *CLOSER (1)*, 2016, pp. 137–146.

[10] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan, "Learn-as-you-go with megh: Efficient live migration of virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1786–1801, 2019.

[11] S. Pal, R. Kumar, L. H. Son, K. Saravanan, M. Abdel-Basset, G. Manogaran, and P. H. Thong, "Novel probabilistic resource migration algorithm for cross-cloud live migration of virtual machines in public cloud," *The Journal of Supercomputing*, vol. 75, no. 9, pp. 5848–5865, 2019.

[12] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 377–390, 2002.

[13] M. Kozuch and M. Satyanarayanan, "Internet suspend/resume," in *Proceedings fourth IEEE workshop on mobile computing systems and applications*. IEEE, 2002, pp. 40–46.

[14] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble, "Constructing services with interposable virtual hardware." in *NSDI*, 2004, pp. 169–182.

[15] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the v-system," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, pp. 2–12, 1985.

[16] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, vol. 43, no. 3, pp. 14–26, 2009.

[17] Z. Lei, E. Sun, S. Chen, J. Wu, and W. Shen, "A novel hybrid-copy algorithm for live migration of virtual machine," *Future Internet*, vol. 9, no. 3, p. 37, 2017.

[18] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[19] Á. Kovács, "Comparison of different linux containers," in *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, 2017, pp. 47–51.

[20] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: a performance evaluation," *Sensors*, vol. 19, no. 7, p. 1488, 2019.

[21] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 85–90.

[22] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove, "Picocenter: Supporting long-lived, mostly-idle applications in cloud environments," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[23] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2137–2142.

[24] H. Dinh-Tuan, F. Beierle, and S. R. Garzon, "Maia: A microservices-based architecture for industrial data analytics," in *2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*. IEEE, 2019, pp. 23–30.

[25] T. Hilgert, M. Kagerbauer, T. Schuster, and C. Becker, "Optimization of individual travel behavior through customized mobility services and their effects on travel demand and transportation systems," *Transportation Research Procedia*, vol. 19, pp. 58–69, 2016.

[26] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in *2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE, 2017, pp. 1–6.

[27] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," *arXiv preprint arXiv:1609.05830*, 2016.

[28] K. Bakshi, "Microservices-based software architecture and approaches," in *2017 IEEE aerospace conference*. IEEE, 2017, pp. 1–8.