MirageManager: Enabling Stateful Migration for Unikernels

Anonymous Author(s)

ABSTRACT

Unikernels are a new lightweight virtualization technology born as an alternative to virtual machines and containers. Geared towards service provisioning for the Internet of Things (IoT) and edge computing, they offer extremely small memory footprint and strong isolation properties. However, the unikernels ecosystem is still in its infancy and lacks quintessential functionalities found in more well-established virtualization technologies. For example, stateful migration is a highly desired feature for mobile edge services in distributed environments which is not yet supported by unikernels. This is one of the shortcomings preventing us from reaping the full benefits of unikernels outside of stateless applications.

In this work, we aim bridging this gap with MirageManager: a ready-to-deploy unikernel migration system enabling lossless migration supported by a function-level, application logic checkpointing library of our design. Our evaluation results show that MirageManager is able to lower the service downtime by ~**80%**, and drastically reduce the state transfer data by almost ~**100%** when comparing against Podman. Additionally, MirageManager also beats Podman, a container-based engine, in parallel service migration across constrained edge networks reducing the overall migration time by up to ~**6x**.

1 INTRODUCTION

In edge computing environments, offloading computation to the nearest edge node is key to cutting network latency and improving user experience [1–5]. However, edge and IoT networks may be unreliable [6] and composed of resource-constrained devices that are more prone to failures. Ideally, the edge infrastructure should be able to self-adapt in case of malfunctions and quickly move the computation to a stable node in order to maintain high service responsiveness and avoid data loss. Therefore, fast service migration and recovery (e.g. reinstantiation) play a crucial role in reducing the overall service downtime.

Migration based on VMs or containers in the edge computing domain has been explored in numerous studies. VM handoff [7] has been proposed to accelerate service handoff across offloading edge nodes. It divided VM images into two stacked overlays based on Virtual Machine (VM) synthesis [8] techniques. In contrast, the wide deployment of containers platforms provides a base for high speed service handover. The Docker storage driver employs layered images inside containers, enabling fast packaging and shipping of any application as a container. Many container platforms, such as OpenVZ [9], LXC [10], and Docker [11], either completely or partially support container migration, but none of them are suitable for the edge computing environment [5]. In fact, LXC migration and Docker migration are based on Checkpoint/Restore In Userspace (CRIU) [12] and need to transfer the whole container file system during the migration, resulting in inefficiency and increasing network overhead as a function of the filesystem size.

To cope with future Internet architectural trends fueled by a pressing need to support edge/fog computing environments, new forms of service decomposition such as lambda functions [13] and unikernels [14] have been developed. While both approaches are designed for stateless applications, we argue that for specific use-cases it is necessary to preserve the execution state. For example, in security applications that collect network traffic data and trigger events if suspicious behavior is detected [15], this data would be lost in a simple image duplication of stateless migration. Time variant computations, such as sensor fusion, would deliver different results depending on when they start or how long they run. In addition, services deployed in radio access networks that are tightly coupled with their mobile user would have to preserve their data in order to stay in sync with their users physically moving from one access point to another [16]. However, unikernel migration has not yet been explored in past research. Most approaches opted for fine-tuned container-based solutions to reduce service downtime at the edge. Meanwhile, we found no tool offering stateful migration for unikernels which, due to their properties (discussed in §2), are a promising option for edge computing applications.

To fill this gap, we developed **MirageManager**: a checkpoint-based, live migration solution for unikernels. Our contribution is two-fold: (i) an architecture and workflow to manage the migration of unikernels and (ii) a library called *Unimem* which can preserve the unikernel state at function level during the migration. Our prototype is based on MirageOS and we strive to abstract from any platform-specific implementation details by handling the execution state explicitly within the application itself before transferring it. Initially geared towards MirageOS, **MirageManager** is designed to be later ported to other unikernel implementations.

The remainder of this paper is structured as follows. We provide background information and review related work in §2. We describe our system design and *checkpointing* technique in §3 and §4, respectively. We discuss the details of our implementation in §5 followed by preliminary results in §6. Finally, we discuss pros and cons of our approach in §7 and conclude the paper in §8.

2 MOTIVATION AND RELATED WORK

We begin by looking at the need for stateful unikernel migration in respect to current trends in the edge computing and IoT domains.

With the rapid development of the edge computing model, many researchers have developed applications exploiting the edge computing paradigm. Machine learning, computer vision and signal processing are just examples of classes of applications that benefit from offloading intensive computation [17–19] to nearby edge devices. Similarly, research exploring service migration for edge computing followed [16, 20–22], highlighting pros and cons of the different approaches adopted. Originally, the idea of shipping and migrating computation was supported by code slicing or VM and container migration [23, 24]. However, these mechanisms require transmission of a considerable amount of data over the network due to sheer size of the execution environment that needs to be migrated. The advantage is portability as less assumptions need to be made regarding the destination machine.

Containers require less data to be transferred as the OS is not migrated during the procedure and are overall more lightweight than VMs [25]. On the other hand, containers depend on OS-specific functions which makes the migration procedure difficult due to the presence of external dependencies [26]. A sweet-spot between these two approaches are *unikernels*: a new, emerging multi-purpose virtualization technology tailored for resource-constrained devices commonly found at the edge [27–29]. Due to their strong isolation properties, reduced attack surface, and small memory footprint, unikernels represent an intersection between containers and traditional VMs. Additionally, their compilation model enables whole-system optimization across device drivers and application logic. In particular, by being in the order of a few MBs in terms of image size, the migration of an unikernel is more resilient to the unfavorable network conditions (e.g., unreliable, intermittent connectivity, limited bandwidth) often found in edge and IoT networks [6]. For example, operations like image or state retransmission are less costly in terms of transmitted data when compared to other virtualization techniques.

Unikernels are still in their infancy and do not offer the suite of functionalities provided by other well-established virtualization tools. This applies also to migration, which, to the best of our knowledge, is still an unexplored path. In fact, unikernels are advertised primarily as stateless appliances which, by definition, do not require migration as no information should be preserved across subsequent executions. However, unikernels have been recently used also in stateful contexts as in [15, 30] where preserving the state of execution is necessary to maintain consistency and avoid gaps in the collected data. Security applications, for example, collect data on network traffic and trigger events if suspicious behavior is detected. For example, in [15] a unikernel based intrusion detection system (IDS) is proposed. In this case, loss of the data structures during service re-instantiation could expose the network to attacks as malicious connection could not be tracked anymore. Additionally, time-variant computations such as sensor fusion will deliver different results depending on when they start or how long they run. Another possible application is motion patterns detection in a video stream, which requires to preserve information from past processed video frames during a migration [31]. However, in this case the unikernel would eventually require access to additional hardware resources (e.g., GPU) for which support is yet not available. For the use-cases mentioned above, stateful migration can help in increasing the service reliability in case of faults and prevent data losses in case of service reinstatiation.

3 SYSTEM DESIGN

Typically, service migration is achieved by dumping the content of a virtual instance into a file and transferring it to a destination host. Then, the hypervisor will take care of restoring the service. However, this procedure also requires support by the guest operating system. While most hypervisors support migration, this is not necessarily the case for all guest OSes. In fact, unikernels do not support it for the reasons explained in §2.

We added the required migration logic directly at the application layer instead of making any changes at the kernel level (e.g., MiniOS [32]) or in the hypervisor. This is a practice followed also in past work for VM-independent migration of stateful applications or to capture the application state at a high-level before migrating it [31, 33]. Hence, we designed a set of functionality in the shape of a library allowing the unikernel to keep track of its own state internally. When the unikernel needs to suspend, it serializes its state so it can be transferred to the migration target, which will process the state before proceeding with the execution flow.

Aside from state tracking (discussed in §4), we require an additional component to support the migration process which we call MirageManager. It is a web service exposing an interface to commission and manage unikernels on any registered host and transfer the unikernel state using a repository. It is the core of our system and it manages the life-cycle (e.g., creation, migration, destruction) of unikernels deployed on multiple hosts and it provides a repository for writing and retrieving the unikernels state before and after a migration. The representation of a unikernel is populated before the hypervisor creates the guest domain and will exist even after its destruction, regardless of whether it is the result of a migration procedure or a regular shutdown. During the guest lifetime, the representation will change to reflect changes in its state.

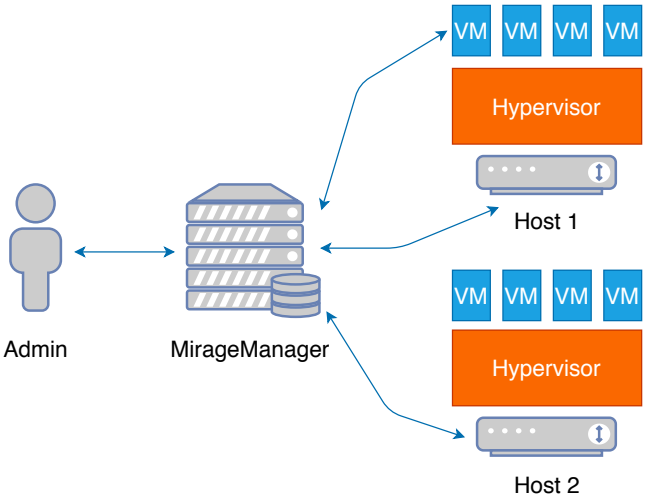

Figure 1: MirageManager.

When a unikernel is started, MirageManager will create a guest domain of the corresponding image on the target host. Afterward, to confirm a successful boot, the unikernel will query MirageManager and start a lookup procedure for a previous state associated to it. This procedure is required so that, even if no state is retrieved, MirageManager will be aware of the current state of the unikernel (specifically, *started*) and change it to *connected*.

At the moment of a migration, MirageManager will issue a *suspend* command to the unikernel. Hence, the latter will transfer its state to the repository and thereby confirm that the suspension was successful. When resumed on the target machine, a state will be retrieved from MirageManager and the guest will use it to update itself before resuming its workflow exactly from where it was interrupted before the migration. This process can be repeated indefinitely until the unikernel is permanently stopped, completes its intended task, or exits due to an error/fault. Potentially, the

unikernel could invoke by itself the migration procedure without the need for an external trigger. However, this functionality is not yet supported in the current version of our system.

4 STATE CHECKPOINTING

In order to manage the lifecycle of a unikernel, MirageManager requires a complete representation of its execution state. Therefore, we developed a module able to store and serialize the unikernel application logic state so that execution can be resumed from it. We call this procedure *checkpointing*, described below.

For the purpose of creating checkpoints, we implement a library for MirageOS that defines a central state object representing the unikernel's state. Additionally, we defined a programming model which allows to express the application logic routines in a serializable format, so that the execution state can be written to the store and transferred to the repository. Such store is called *Unimem* and it is implemented as a key-value store using strings as keys and a polymorphic data-type for the values. The currently supported data-types are single element, list, or list of lists. The application can decide to write either variables (e.g., intermediate execution results) or details about its execution state into Unimem, depending on the context. Additionally, Unimem also encapsulates the communication protocol semantics used to dialogue with MirageManager, which are shown in Figure 5.

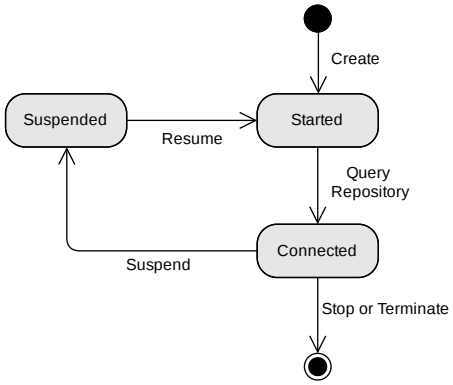

Figure 2: MirageManager — Unikernel lifecycle.

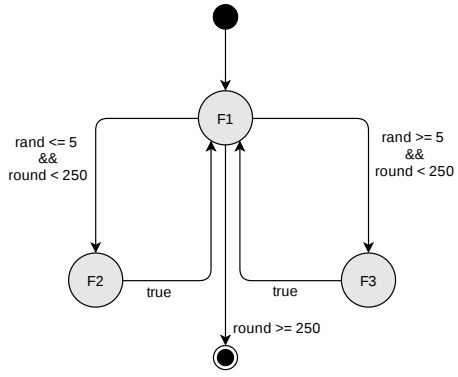

Figure 3: Checkpointing execution graph.

To enable the application to write its execution state to Unimem, we translate the unikernel into a series of atomic procedures, each constituting a step. Hence, we define the application workflow as a directed graph with labeled edges where each node is a computational step. Every step is identified by a unique string identifier (ID) so that the currently active step can be dumped into the store just by using its ID. Edges are guarded by expressions using the variables present in the store that determine how the control flow is directed from one step to the next.

In the case of multiple edges originating from the same computational step (e.g., execution logic branching), the program decides which one to follow by evaluating what we call *transition guards* which are conditional equations evaluated on variables stored in Unimem. Therefore, the control flow can be expressed as an adjacency matrix where each entry a_{ij} describes the transition from step *i-th* to *j-th* and its value acts as a guard for the transition. The truth condition of the transition guard is obtained from evaluating specific functions on a set of variables in the store. The first condition evaluating to *true* in a row of the adjacency matrix determines the next transition in the execution flow. One limitation is that guard functions should be mutually exclusive to avoid ambiguities in the process of selecting which transition to take at any given moment. If no guard condition evaluates to *true*, the application logic is considered to be completed and the unikernel terminates.

When the unikernel is requested to suspend or migrate, the identifier of the currently executed function is written to Unimem. As every variable used to evaluate the transition guards is stored as well, Unimem's content fully describes the application state. In fact, the current position in the graph as well as the next transition to be traversed can be inferred from the stores content only. To protect against state corruption and potential information loss, all variables belonging to an execution scope spanning multiple computational steps must be stored. This is facilitated by not using return values or parameters for the steps, but rather writing from and reading to the store. As computational steps are atomic, the information contained in Unimem is sufficient to recreate the application state after a migration.

Finally, there is no specific structure imposed on the content of Unimem by MirageManager as long as the state is serializable. Therefore, also other state information, such as the state of an object-oriented application, could theoretically be stored.

5 IMPLEMENTATION

MirageManager is implemented as a distributed system consisting of an application server developed with Express [34] and written in JavaScript. It exposes a REST API for the admin user to issue migration commands, and for the unikernel to transfer its state to the central repository. Additionally, each host wishing to use MirageManager needs to run a controller so that the central application server can communicate with the hypervisor on that machine. In Figure 4, the *ServerNode* hosts the application server while the *EdgeNode* is as device using the migration functionalities. In our implementation, we used Xen as hypervisor [35] but other options are possible, as discussed in §7.

The communication between application server and host controller is performed by remote procedure calls using gRPC [36].

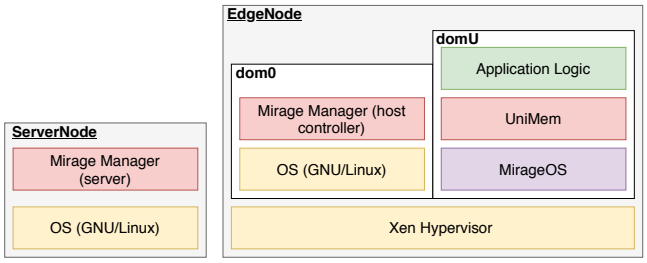

Figure 4: MirageManager components.

The controller programmatically issues commands to Xen via the *xl* tool in order to create and destroy domains. Additionally, it uses *xenstore-write* to communicate with the unikernel guest domains. The application server API is invoked using HTTP requests for issuing commands and transferring states which are encoded using the JSON format.

Inside the unikernel, the state is stored in a Unimem object which is a singleton instantiated from an object class in our library's store module. At the core of Unimem there is a key-value store implemented using the OCaml Map module. We expressed the adjacency matrix so that the keys are the origin compute step IDs and the values are a list of records containing the tuple destination node ID and transition guard. The OCaml code snippet in Listing 1 shows the implementation of the execution flow in Figure 3.

Unimem also embeds the communication protocol functions to communicate with MirageManager and serialize the unikernel state.

```
let get_adjacency store =
  let g12 = (((Store.to_int
    (store#get "rand" (Store.VInt 0)))>= 5)
   && ((Store.to_int
    (store#get "round" (Store.VInt 0)))<=250)) in
  let g13 = (((Store.to_int
    (store#get "rand" (Store.VInt 0))) < 5)
   && ((Store.to_int
    (store#get "round" (Store.VInt 0))) <=250)) in
  let gt = ((Store.to_int
   (store#get "round" (Store.VInt 0))) > 250) in
  let assoc_adj_list = [
    ("f1", [
      {step = "f2"; condition = f12};
      {step = "f3"; condition = f13};
      {step = "terminate"; condition = gt};
    ]);
    ("f2", [{step = "f1"; condition = true}]);
    ("f3", [{step = "f1"; condition = true}]);
  ] in
  let amap = StringMap.of_seq (List.to_seq
   assoc_adj_list) in
  amap
```

Listing 1: Unimem code snippet.

In addition to the Unimem class, there are utility functions for converting the store value types to normal OCaml types in the store module. Finally, the library offers a control module providing a set of functionalities allowing the unikernel to communicate with the controller through Xenstore [37].

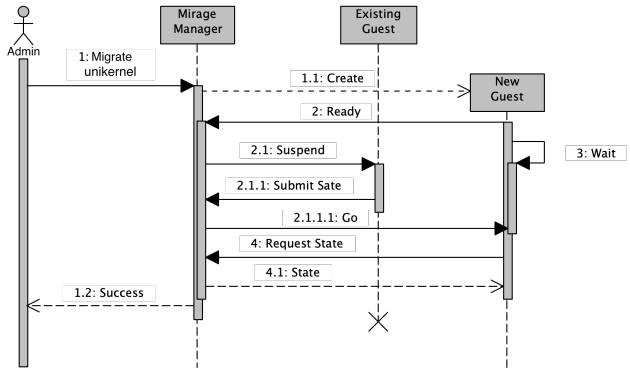

Figure 5: MirageManager migration workflow.

6 EVALUATION

For our evaluation, we selected Podman as candidate baseline to compare against MirageManager. Podman is an engine for running Open Container Initiative (OCI) containers with support for CRIU-based migration for Docker. We embedded an application with the same functionalities as our MirageOS unikernel inside an OCI container and then performed the migration tests. The application consisted of a simple numeric counter printing the current number of iterations, at every second. As we focus on the IoT domain, we decide to keep the application logic simple. At this stage, we did not consider the possibility to run and embed complex applications in an unikernel running on a contained device. We did not compare MirageManager against Docker's native container migration tool because it is still in an experimental stage and required excessive modifications and workarounds to use it in our tests. Therefore, we excluded it from our evaluation.

The migrations operations were performed between two Intel NUCs connected to the same subnet with a 100Mbps connection and running Ubuntu 18.04 with a downgraded kernel version (due to incompatibilities with CRIU). To provide a quantitative analysis of our solutions against Podman, we selected four metrics:

Downtime. The time elapsed between service suspension and restart. It is the most critical metric for evaluating the performance of a live migration, as it shows for how long the service is not able to perform its task. From a user perspective, only the downtime is noticeable. As timestamps are logged for every iteration of the unikernel application logic, we can precisely measure the downtime by subtracting the two timestamps between suspension and restart.

Migration Time. It is the time required by MirageManager to perform the state migration of a unikernel between source and target machine, including the resume operation. Compared to downtime, it is a compound metric covering different steps of our migration workflow and it is calculated differently for Podman and MirageManager. For the former, a timestamp is logged both when the suspend command is issued and when Podman completes the resume command. The difference between the two values amounts to the migration time. For the latter, the first timestamp is generated at suspension time, but the second is created by the unikernel after successfully retrieving its state from the repository. We can breakdown this time interval even further. The *init* time tantamount to

the kernel boot plus the initialization of a TCP/IP network connection. During the *wait* time, the new unikernel awaits for the old one to suspend and save its state. Finally, the *retrieval* time represents the time interval to query the state from the repository. These three phases show the proceeding in time of the migration workflow shown in Figure 5.

State Size. Amounts to the data needed to be transferred between hosts in order to perform the migration. This metric is less crucial for migrations happening in datacenters where bandwidth is not an issue. However, as we focus on migration in edge networks, it assumes much more relevance as the available bandwidth is limited. The state size is calculated in Bytes and for Podman is the size of the state tarball while, for MirageManager, of the JSON message embedding the state of the unikernel.

Image Size. The size of the image. Both, for Podman and MirageManager it is defined by the size of the image stored in the filesystem.

Results

Table 1 shows the migration results as the average of 200 migrations. MirageManager provides slightly worse performance, as it takes longer than a cold migration with Podman. However, this is due to MirageManager already starting the target unikernel before transferring its state over the network. This is a time consuming operation due to the time required for MirageOS to successfully setup the network channel, especially with DHCP enabled.

Table 1: MirageManager vs. Podman

		MirageManager	Podman
Migration Time	Init [s]	0.91 +/- 0.20	-
	Wait [s]	2.61 +/- 0.45	-
	Retrieve [s]	0.02 +/- 0.006	-
	Total [s]	3.54 +/- 0.48	1.96 +/- 0.06
	Downtime [s]	0.33 +/- 0.02	1.80 +/- 0.05
	State Size [B]	79.00	195175.28 +/- 113.39
	Image Size [B]	27780862	70730752

The benefit of our approach can clearly be seen in terms of downtime. In fact, MirageManager downtime is ~**80%** shorter than Podman's. Regarding data usage, MirageManager is clearly in advantage. The container migration requires a full memory dump transfer, while MirageManager only transfers the necessary variables to preserve the application logic state. As a consequence, the amount of data needed to transfer the state between machines is more than ~**2000x** smaller with MirageManager, when compared to Podman. This is also a function of the application logic which can affect the state transfer *cost*.

An important factor when evaluating service migration in distributed systems is scalability. Therefore, we evaluated how migration with MirageManager fares in comparison with Podman when both tools perform multiple migrations simultaneously and measured the overall migration time. Figure 6 shows the overall time required to migrate multiple services in parallel. The top part of the plot shows the results for MirageManager while the bottom part shows those for Podman. In both cases, we measured the overall migration time with four different bandwidth settings.

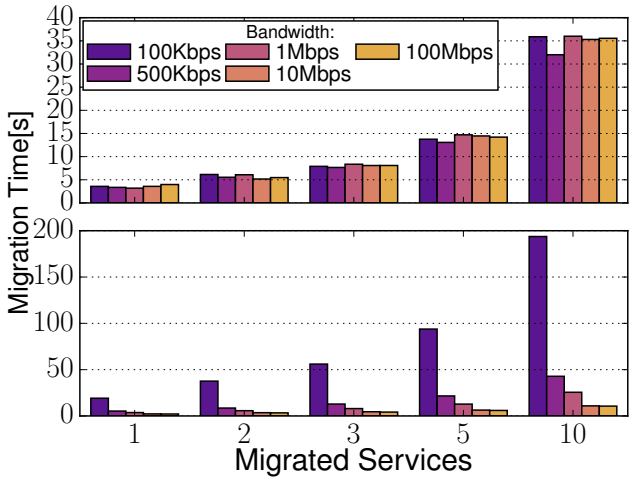

Figure 6: Migration scaling performance. MirageManager (top) vs. Podman (bottom).

As we discussed previously, edge networks suffer from bandwidth constraints which severely impact migration operations when the transferred state is not small. This stresses the need not only to follow best-practices of service decomposition but also to reduce the state size as much as possible. For both, unikernels can be the answer.

We can gain multiple insights from Figure 6. First, MirageManager's migration time is seemingly unaffected by the available network bandwidth and it grows quasi-linearly with the amount of services migrated in parallel. The transferred state is extremely small, as we do not include the domains full memory. The same cannot be said for Podman, which is definitely suffering in low bandwidth conditions because it needs to transfer the complete memory dump as part of its migration technique. This tendency is exacerbated with the network bandwidth capped at 100 and 500 Kbps. In this case, MirageManager is up to ~**6x** times faster than Podman. On the other hand, Podman outshines MirageManager as the available bandwidth increases. In fact, the latter it heavily penalized by the long *wait* time (as shown in Table 1) which is the major culprit of the long migration time. However, this is a limitation of the specific unikernel rather than our system which can be addressed in the future to drastically improve MirageManager performance.

Finally, while migration with Podman is transparent to the migrated application, MirageManager requires changes to the application logic in order to work correctly. Based on this, we state that MirageManager generally outperforms in downtime and data transfer volume cold migration with containers while offering competitive performance in terms of overall migration time.

7 DISCUSSION

In this section we discuss the limitation of our approach in relation to the our implementation and design choices.

MirageOS & OCaml. Currently, MirageManager only supports MirageOS unikernels. While MirageOS is a promising project, this

results in MirageManagers biggest limitation as it forces the developer to write all code in a specific programming language (OCaml). Additionally, MirageOS unikernels compiled against Xen do not support the full set of libraries available to POSIX processes. This is due to the restricted set of libraries that have been ported to be compatible with MiniOS. However, our system could be extended and ported to work with other unikernels [38–41], which would bring more freedom in terms of available programming languages.

Virtualization. MirageManager uses Xen as hypervisor. However, in recent years we noticed how more flexible and user-friendly solutions, such as KVM [42], have received increasing attention. MirageOS is compatible with KVM and especially Solo5 [43]: a sandboxed execution environment for unikernels based on KVM. Our system could be adapted to run on top of this hypervisor, too, which would also drop some stringent requirements inherited from Xen in terms of, for example, hardware prerequisites.

Application Design. MirageManager imposes further design and implementation restrictions on a newly developed unikernel. The developer must build the application logic so that it can be serialized for a migration. This adds complexity to the development phase and requires specific knowledge of the underlying migration system. On the other hand, MirageOS unikernels benefit from a compile-time defined behavior which opens to the possibility of programmatically generating the adjacency matrix representing the execution flow. Formal proof management system like Coq [44] are natively compatible with OCaml and can help in this regard. Alternatively, we contemplate the possibility of using tools such as pre-processors in order to cope with the code modifications and language implications (e.g. return values) discussed earlier.

While these restrictions can rule out using MirageManager in some cases — what we presented is an initial prototype. Still, it is the first system enabling the migration of unikernels while managing multiple Xen hosts and their guest domains. Our design allows to easily extend the implementation to accommodate diverse hypervisors and library operating systems and, yet at an early stage, it performs competitively when compared to more mature solutions.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented MirageManager: a checkpoint-based, live migration solution for unikernels. We discussed the motivation and reasoning behind our design which stems by a surging interest for service migration at the edge. In order for unikernels to keep growing as a virtualization technology, functionalities like migration must be made available in order to extend their applicability also to stateful services. MirageManager was developed on top of MirageOS and Xen. Our evaluation showed the potential of our solution in comparison to a well established service migration approach. Nevertheless, there are limitations which open to manifold exploration paths for our future work. Improving scalability, reducing the implementation effort by automatically extracting the execution flow, testing our solution with other unikernel technologies are the first challenges we plan on tackling.

REFERENCES

[1] Brandon Amos, Bartosz Ludwiczuk, Mahadev Satyanarayanan, et al. Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, 6:2, 2016.

[2] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010.

[3] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.

[4] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13. IEEE, 2016.

[5] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[6] Yuang Chen and Thomas Kunz. Performance evaluation of iot protocols under a constrained wireless access network. In *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, pages 1–7. IEEE, 2016.

[7] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive VM handoff across cloudlets. *Technical Report CMU-CS-15-113*, 2015.

[8] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[9] OpenVZ. https://wiki.openvz.org/Main_Page. Accessed: 2020-06-16.

[10] LXC. https://linuxcontainers.org/. Accessed: 2020-06-16.

[11] R Boucher. Live migration using CRIU, 2017. Accessed: 2020-06-16.

[12] CRIU. https://criu.org/Main_Page. Accessed: 2020-06-16.

[13] Aws lambda. https://aws.amazon.com/de/lambda/. Accessed: 2020-09-16.

[14] Michał Król and Ioannis Psaras. Nfaas: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 134–144, 2017.

[15] Vittorio Cozzolino, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. UIDS: Unikernel-based Intrusion Detection System for the Internet of Things. In *DISS 2020 - Workshop on Decentralized IoT Systems and Security*, 2020.

[16] Shangguang Wang, Jinliang Xu, Ning Zhang, and Liu Yujiong. A survey on service migration in mobile edge computing. *IEEE Access*, PP:1–1, 04 2018.

[17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[18] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78. IEEE, 2015.

[19] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42, 2015.

[20] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container migration in the fog: a performance evaluation. *Sensors*, 19(7):1488, 2019.

[21] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. Virtualization and migration at the network edge: An overview. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 368–374. IEEE, 2018.

[22] Paolo Bellavista, Alessandro Zanni, and Michele Solimando. A migration-enhanced edge computing support for mobile devices in hostile environments. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 957–962. IEEE, 2017.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[24] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[25] Flávio Ramalho and Augusto Neto. Virtualization at the network edge: A performance comparison. In *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6. IEEE, 2016.

[26] Motoshi Horii, Yuji Kojima, and Kenichi Fukuda. Stateful process migration for edge computing applications. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[27] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, 2018.

[28] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, et al. Personal data management with the databox: What's inside the box? In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 49–54, 2016.

[29] Anil Madhavapeddy and David J Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[30] Vittorio Cozzolino, Jörg Ott, Aaron Yi Ding, and Richard Mortier. Ecco: Edge-cloud chaining and orchestration framework for road context assessment. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 223–230. IEEE, 2020.

[31] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[32] MiniOS. https://wiki.xenproject.org/wiki/Mini-OS. Accessed: 2020-06-16.

[33] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269, 2016.

[34] Express framework. Accessed: 2020-06-16.

[35] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[36] GRPC framework. Accessed: 2020-06-16.

[37] Xenstore-write manual. Accessed: 2020-06-16.

[38] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[39] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 459–473, 2014.

[40] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 61–72, 2014.

[41] rumprun. https://github.com/rumpkernel/rumprun. Accessed: 2020-06-16.

[42] Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page. Accessed: 2020-06-22.

[43] Solo5. https://github.com/Solo5/solo5. Accessed: 2020-06-16.

[44] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.