# Live Service Migration in Mobile Edge Clouds

Andrew Machen, Shiqiang Wang, Kin K. Leung, Bong Jun Ko and Theodoros Salonidis

## Abstract

Mobile edge clouds (MECs) bring the benefits of the cloud closer to the user, by installing small cloud infrastructures at the network edge. This enables a new breed of real-time applications, such as instantaneous object recognition and safety assistance in intelligent transportation systems, that require very low latency. One key issue that comes with proximity is how to ensure that users always receive good performance as they move across different locations. Migrating services between MECs is seen as the means to achieve this. This article presents a layered framework for migrating active service applications that are encapsulated either in virtual machines (VMs) or containers. This layering approach allows a substantial reduction in service downtime. The framework is easy to implement using readily available technologies, and one of its key advantages is that it supports containers, which is a promising emerging technology that offers tangible benefits over VMs. The migration performance of various real applications is evaluated by experiments under the presented framework. Insights drawn from the experimentation results are discussed.

## Introduction

Cloud-based mobile applications have become increasingly popular in recent years [1]. One key issue is to ensure that services are always delivered with good performance. The current centralized structure of the cloud has led to a generally large geographical separation between the users and the cloud infrastructure. In such a setting, end-to-end communication between user and cloud can involve many network hops, resulting in high latency. The ingress bandwidth to the cloud may also suffer from saturation as the cloud infrastructure is accessed on a many-to-one basis [2].

A promising approach for resolving the above problems is to install computing infrastructures at the network edge. Particularly for real-time applications such as instantaneous object recognition [3] and safety assistance in intelligent transportation systems (ITS) [4], service applications have to remain in relatively close proximity to their end users in order to ensure low latency and high bandwidth connectivity. This is captured by the newly emerged concept of mobile edge clouds (MECs) [5], as well as similar concepts such as cloudlet [2], fog computing [4], follow-me cloud [6], mobile micro-cloud [7], and small cell cloud [8]. While these different concepts may propose slightly different implementations, they all suggest placing small cloud infrastructures at the network edge so that users can have seamless connection to cloud services. In particular, MECs are typically placed only one or a few network hops away from the mobile user, thus the communication latency can be kept very low. The ingress traffic into the backhaul network can also be reduced, because large amounts of data can be processed directly at the edge. It is envisioned that MECs will co-exist and work in unison with the existing centralized cloud that we have today, as shown in Fig. 1.

To maintain the benefits of running services close to the user, when a mobile user moves away from its original location, its service may need to be migrated to a new MEC server that is near the current user location [6]. This article focuses on systems aspects of live service migration in MEC environments. The main challenge is how to maintain relatively low service downtime and overall migration time. We address this challenge with a layered framework that decomposes a cloud application into multiple layers so that only those layers that are missing at the destination are transferred. Our framework applies to both virtual machines and containers and can be readily implemented with existing tools. To the best of our knowledge, this is the first systematic study of live migration of MEC applications in a container-based environment.

In the rest of this article, we first summarize the motivation and background, then propose our layered framework for live service migration and its experimentation results.

## Live Service Migration: Motivation and Background

The need for live service migration in MECs can be illustrated with the example shown in Fig. 1. Here, mobile user 2 is initially connected to its service running on MEC 1. The direct connection between user 2 and MEC 1 ensures low-latency access to the service. However, after some time, user 2 moves to a location that does not have direct connection to MEC 1; it has direct

Andrew Machen was with Imperial College London and IBM U.K. when this work was performed; Shiqiang Wang, Bong Jun Ko, and Theodoros Salonidis are with IBM T. J. Watson Research Center; Kin K. Leung is with Imperial College London; The corresponding author is S. Wang.

connection to MEC 2 instead. As shown in [9], connecting to MEC 1 in such cases would incur a significantly higher latency (due to backhaul network communication) than connecting to MEC 2 directly. It can therefore be beneficial to migrate user 2's service from MEC 1 to MEC 2, so that user 2 can continue enjoying low-latency access to its service. Such live migrations may be frequently required, especially in vehicular applications where it is likely that the users have high mobility.

Service migration can be classified into stateful and stateless migrations. Stateless migration does not move application running states, it only redirects the user requests to a new server with a separate instance of the service running. This is applicable for applications that do not keep states for users. However, for interactive services that are becoming increasingly popular today, such as active safety warning, mobile multimedia, and mobile online gaming, it is very likely that the application needs to keep some state for each user. We therefore focus on stateful migration in this article, which involves moving running states of applications. We consider the stateful migration of a guest operating system (OS) hosting service applications, where a user receives service for a continuous period of time, and the service application may need to keep some internal state for the user (e.g., some intermediate data processing results). After migration is completed, programs resume exactly where they left off before migration, thus the migration is classified as live. The user starts to receive service before migration occurs, and it continues receiving service after migration.

**Optimization of Migration Decisions:** Migration can incur service interruption as well as computation and communication resource overheads. Therefore, the decision on whether, when, and where to migrate depends on many aspects, such as user mobility, communication channel characteristics, resource availability at MECs, and so on, which is a sophisticated optimization problem. In essence, there is a trade-off between the cost of migration and the benefit after migration. Algorithms for making migration decisions need to balance this trade-off. They usually need to predict the future service demands with some accuracy, or buffer service requests in queues so they can be served in batches possibly after migration. Readers are referred to [7, 10] on how live migration decisions can be formulated as optimization problems and how to solve these problems.

**Execution of Live Migration:** An active systems research challenge is how to efficiently execute live migration in a practical cloud system containing MECs. We first recall that services running on a cloud platform are likely to have OS and application dependencies that need to be met by their host system. Therefore, a service application is often encapsulated into its own self-sufficient and pre-configured environment for easy distribution. Current examples of such an environment are the well established hypervisor-based virtual machines (VMs), or the relatively new technique, containers. Both technologies allow the creation and running of multiple isolated guest OSs on top of a host OS. The main difference between
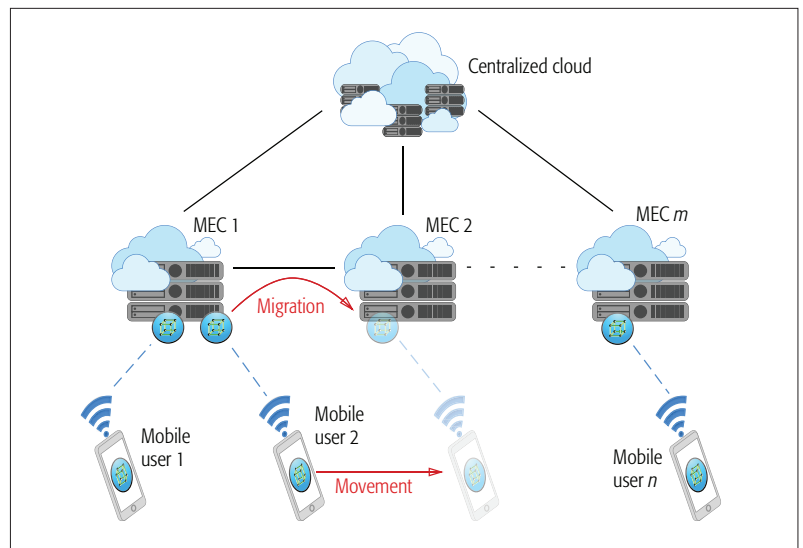


**FIGURE 1.** Mobile edge cloud (MEC) architecture.

the two technologies is that VMs fully emulate the OS kernel and hardware, whereas containers directly share the hardware and kernel with their host machines. As a result, containers occupy much less resources and have lower virtualization overhead than VMs, but are less adaptable, e.g., a Linux container cannot run on a Windows server.

Recent efforts toward the implementation of service migration in MEC environments has focused on VM migration [9]. Container migration is a relatively new area that has not been systematically studied in the literature. As containers usually have a much smaller size than VMs, it can be very beneficial to run container-based applications on MECs that have limited storage and processing capability. Thus, a natural question to ask is how to support the live migration of containers. We particularly would like to support container live migration without drastically changing the existing container implementations, so that minimal effort is required to add this functionality to existing systems.

We should also note that there are existing VM live migration methods for cloud environments [11]. However, most of them are built for data centers, requiring the use of storage area networks (SANs) and shared storage. Moreover, these methods are usually specific to the underlying virtualization technology. The method presented in this article is designed to work over wide area networks (WANs), which are envisioned to be how MECs will be interconnected, and it is a generic mechanism that applies to different types of containers and VMs.

## A Layered Framework for Live Migration

We present a generic layered migration framework using incremental file synchronization, *which works with both containers and VMs*. A benefit of this framework is that it is built based on readily available functionalities in most container and VM technologies that are popular today, which means that one does not need to modify the internals of the container and VM implementation in order to apply this framework.
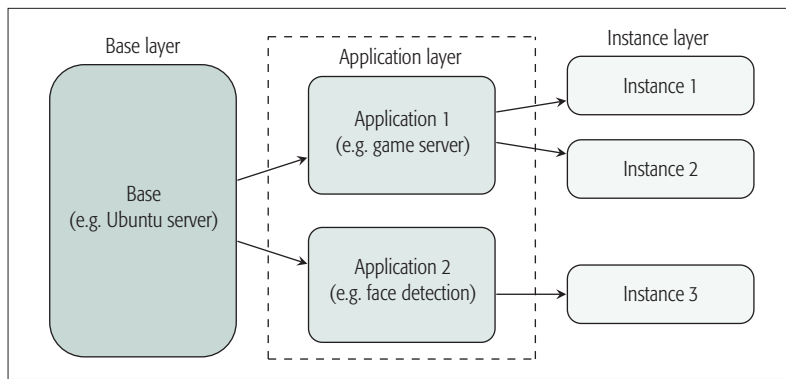
FIGURE 2. The three-layer model, where the dashed box outlines the additional building block (i.e., the application layer) that does not exist in the two layer model.

We focus on LXC (linuxcontainers.org) and KVM (www.linux-kvm.org) as representative technologies for containers and virtual machines, respectively. LXC and KVM are chosen for their popularity, and their ability to run Linux-built applications without modification. We note that this article focuses on the migration of back-end application components between different MECs, where LXC and KVM are applicable because the server often has a Linux-based operating system. The core idea of the method we use for live migration is derived from [12], which proposed an LXC live migration mechanism. We have largely extended [12] so that our approach works with multiple layers (see below), and applies (with minor alterations) to the live migration of KVM and undoubtedly other container and VM technologies as well.

In the following, we first describe a basic procedure that we have developed for performing stateful live migrations in MEC environments. Then we describe our layered framework built on top of this basic procedure that optimizes live migration time further.

## Basic Procedure of Stateful Live Migration in MECs

To migrate an application, the in-memory state of a running guest OS is recorded, transferred, and then recreated at the destination. The in-memory state includes the applications, system processes, and resources currently loaded into memory for quick access. It contains the progress (state) of running applications, including any data that the application is currently working on. Transferring the in-memory state makes it possible to restore an application exactly from where it was suspended.

The migration framework we present uses accessible tools that already exist in container and VM technology. In LXC, this goes by the name of *checkpointing*, and in KVM, *saving*. Both methods suspend the guest OS (at which point the service is temporarily stopped) and save down the in-memory state of the guest OS into one or more files that can be easily transferred. Complementary tools exist to restore the guest OS from their checkpoint (or save).

After the in-memory state is saved into files, the next step is to use a file transfer protocol to transfer the guest OS's filesystem (i.e., all files saved on the hard disk of the guest OS) and saved

in-memory state.[1] For this we use incremental file synchronization, namely in the form of *rsync* – a well-known file syncing tool (rsync.samba.org), to compress and transfer files. A major difference of incremental file synchronization over other basic file transfer protocols is its ability to identify and transfer only those files, or parts thereof, that are different from those already located at the destination. This can substantially reduce the amount of data that needs to be transferred, particularly with our layered framework presented next.

### Layering

The problem with migrating an encapsulated service directly using the above basic live migration procedure is that the "package" contains the guest OS, virtualization data, and, for VMs, the system kernel, which are required to make the service self-contained. A base package with no services installed can have a size that tips the scale at about 400 MB for LXC and 2.7 GB for KVM. Our experiments using the live migration method discussed above have shown that, for a 100 Mb/s bandwidth connection, the average migration time for a base package is 25 seconds for LXC and 160 seconds for KVM, during which time the service is down and may appear as an unresponsive or frozen application.

In our approach, we aim to reduce service downtime (i.e., time of service interruption) and overall migration time (i.e., time from the beginning until the end of the whole migration process) through the use of layers. Abstractly, we can separate the base package (that includes the guest OS, kernel, and so on, but with no applications installed) into a layer separate and distinct from its service applications. We call this package the *base layer*. This base layer is generic and all MECs shall have a copy of it. The service applications and their running states are placed within a separate layer called the *instance layer*. Assuming the base layer exists at the destination, when we want to migrate a service, we first suspend the service and then transfer only the instance layer to the destination. A running service can be reconstructed from a combination of the base and instance layers. By removing the need to transfer the base package in every migration, we can drastically reduce the amount of data that needs to be transferred, and in turn reduce the service downtime and overall migration time.

We extend this *two-layer* approach further, by splitting out the application from the instance layer into an intermediate *application layer* that contains an idle version of the application and any application-specific data. The instance layer now only needs to contain the running state (i.e., in-memory state) of a service. When we want to migrate a service, the application layer can be migrated first while the service is running, then the service is suspended and the instance layer is transferred to the destination. A running service can be reconstructed from a combination of the base layer, instance layer, and associated application layer. By doing this, we are able to transfer the majority of an encapsulated service's program and data before suspending the service, leaving only the running state transfer to count toward the service downtime. This three-layer setup is shown in Fig. 2, and is referred to as the *three-layer model*.

[1] Note that besides the filesystem and in-memory state, we also need to transfer any additional files/data related to the container or VM virtualization itself. We do not specifically discuss those additional portions of data, and only emphasize on the main part of data being transferred. Also, the filesystem and in-memory state may be saved into a single file (in the case of a VM image) or multiple files (in the case of containers). We do not separately discuss them since rsync can find differences in two single files as well as multiple files.

The intermediate application layer of the three-layer model has advantages other than improvements to service downtime. A distinct application layer allows the application and its related data to be distributed independently of a running service. Through smart pre-distribution of services, possibly by application caching based on demand prediction, the overall migration time can be reduced to only the time it takes to migrate the instance layer.

In the current implementation, a layer contains its unique data (including files and possibly in-memory state) plus all the data from its preceding layers. We achieve pseudo-incremental layering for migration, by first cloning a lower layer (e.g., application layer), and then using incremental file synchronization to transfer only the difference between that and the higher layer (e.g., instance layer) we want to recreate. This process of cloning and incremental file synchronization is repeated depending on which layers are missing at the destination. The complete migration process is shown in Fig. 3. The time to start the migration and the destination of migration can be determined by an optimization algorithm (see [7, 10] for details).

## EXPERIMENTATION RESULTS AND DISCUSSIONS

We study the live migration performance of our layered framework for a variety of applications. We ran experiments in three "host" VMs connected to an emulation framework, where the emulation framework is the open-source Common Open Research Emulator (CORE) [13]. The host VMs were each given two virtual CPU cores and 2 GB of virtual memory from a physical machine with 2.6 GHz Intel Core i7 and a total of 16 GB 1600 MHz DDR3 memory. Two host VMs acted as MECs, between which migration was carried out, and the third host VM acted as a user requiring MEC service. Unless otherwise specified, the bandwidth between MECs was configured as 100 Mb/s with only system-inherited latency and jitter. The connection between each MEC and the user was configured with 100 Mb/s bandwidth, 25 ms latency, and 5 ms jitter. These connection specifications are typical for wide-area wired networks and local wireless networks, on which MECs and their users operate. They are much more inferior than network connections in data centers. Migration times were calculated by a migration script at each stage of the migration process. We use rsync for incremental file synchronization and the amount of transferred data was measured using Wireshark.

Nested KVM and LXC, which contain running applications, were run inside the two host VMs that mimic MECs. The guest OS of the nested KVM and LXC was Ubuntu 15.10. Note that the base OS image sizes for KVM and LXC are different due to their different virtualization mechanisms, but this has no bearing on our results since, in our experiments, the base layer resides in every MEC and is not included in the migration. As we see below, the installation footprint of applications can be larger for LXC than for KVM, because the Ubuntu installation of LXC has fewer packages included as standard. As such, the instal-
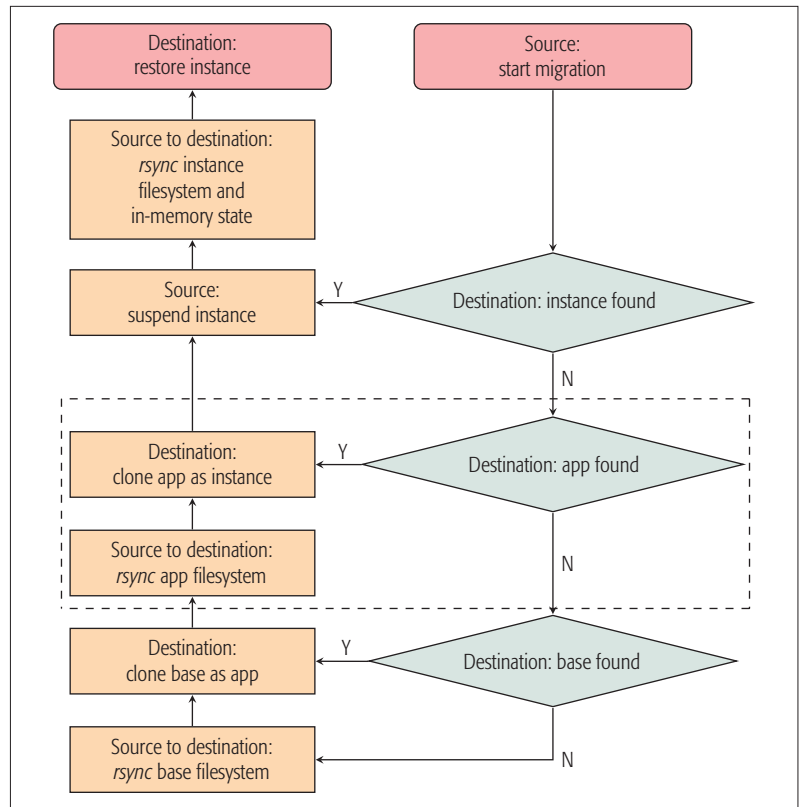


**FIGURE 3.** Flow chart of the migration mechanism for live-migrating with the three-layer model, where rsync is used for incremental file synchronization. The dashed box outlines the additional building block (i.e., operations related to the application layer) that does not exist in the two layer model.

lation of any missing packages is counted into the LXC application size.

We studied the migration of the following applications using our migration framework presented above.

**Game Server:** Runs the sauerbraten-server package, a server for the online game Cube 2. It sends and receives regular packets related to player location and other match statistics. The installation footprint is very small (0.7 MB), as is the memory requirement (approximately 1 MB).

**RAM Simulation:** A simple script (approximately 0.1 MB) that consumes a large amount of RAM, where the exact RAM consumption is user-defined and the RAM contents keep changing over time. It represents memory-intensive applications, such as those that process large data sets or perform complex calculations (e.g., big data analysis, training of deep neural networks, and so on). Unless otherwise specified, RAM utilization is maintained at around 330 MB.

**Video Streaming:** Uses the vlc-nox package (approximately 280 MB for LXC, 230 MB for KVM) to stream video to a user. A 50 MB video file is stored with the application at the MEC. Video streaming has a low memory requirement (approximately 30 MB).

**Face Detection:** Uses the OpenCV library to process an incoming video stream. It detects faces in the video received, and sends the detection result, embedded into each video frame, back to the user. This application has a very large installation footprint (approximately 655 MB for LXC, 565 MB for KVM), and a moderate memory

requirement of approximately 100 MB.

**No Application:** Is a guest OS with no applications installed, and therefore requires no additional resources. It represents the overhead, or minimum bound, on migration.

The experimentation results (averaged over 10 independent experiments in each case) are shown in Table 1 and Figs. 4 and 5. For a particular setup, the service downtime is always smaller than the total migration time, because the total migration time also includes the time taken for cloning and data transfer (during which the service remains running). The time required for different migration stages are unequal for different applications, as shown in Fig. 4, due to the different installation and RAM occupation sizes. This aligns with the layered abstraction discussed earlier. For example, the time required for rsyncing the application filesystem is related to the application installation size, and the time required for rsyncing the instance in-memory state is related to the application's RAM usage.

## CONTAINER VS. VIRTUAL MACHINE

Although it is understood that containers are more lightweight than VMs, a quantitative view on the difference in their migration performance does not exist in the literature. The container-supported migration framework presented above allows us to provide such a quantitative view and draw further insights.

We can see from the experimentation results that for all the example applications shown, LXC has a clear advantage over KVM in terms of total migration time, service downtime, and amount of transferred data. This is mainly because containers are more compact and the filesystem and in-memory contents of a container are mostly relevant to the application, whereas the filesystem and in-memory contents of a VM can be related to many other background processes irrelevant to the considered application, and rsync (or any other incremental file synchronization mechanism) needs to remotely compare a larger amount of data and may not be able to filter out everything that does not belong to the application.

From Fig. 5 (left), we see that for the RAM Simulation application, the migration times of both LXC and KVM are approximately linear in RAM utilization. This linear relationship is because the amount of in-memory state data that needs to be transferred is proportional to the RAM usage. The lower bound on RAM usage represents the overhead required to migrate LXC or KVM. Depending on the application's RAM utilization, we can migrate approximately up to 19x the number of containers compared to VMs within the same amount of time. That said, the relative advantage of LXC over KVM is reduced as the RAM utilization increases. At 400 MB of RAM usage, LXC is 5x faster to migrate than KVM; whereas at 600 MB of RAM usage, LXC is only 4x faster to migrate than KVM.

Figure 5 (right) shows that the relative advantage of LXC over KVM increases as bandwidth increases. At 1 Mb/s, LXC is 2x faster to migrate than KVM, whereas at 10 Mb/s LXC is 4x faster. We found in our experiments that the data transmission rate is capped by how fast rsync is able to compare files and compress data. When the bandwidth is over 50 Mb/s, the migration time remains about the same, where LXC is 8x faster to migrate than KVM.

However, containers also have their shortcomings. By relying on the host system for both the hardware and kernel, they are less adaptable than VMs. They can be nested within VMs to regain their adaptability, but this is likely to degrade performance. Instead, it is advisable that MEC infrastructure is chosen to ensure container compatibility, in as much as the OS (most likely a Linux derivative) remains consistent across the MEC network.

There are also considerations other than performance that will need to be addressed, notably security, in order for containers to be recommended over VMs in all scenarios. Security, a pre-requisite for any enterprise software, is a serious concern with containers, as unlike their VM counterparts, they are not fully isolated from their

| LXC total migration time | Two layers | Three layers (app not found) | Three layers (app found) | LXC service downtime | |
| --- | --- | --- | --- | --- | --- |
| LXC total data transferred | | | | Two layers | Three layers |
| No application | 6.5 s | 11.0 s | 6.3 s | 2.0 s | 2.0 s |
| | 1.4 MB | 1.9 MB | 1.4 MB | | |
| Game server | 7.3 s | 10.9 s | 6.4 s | 3.0 s | 2.0 s |
| | 2.2 MB | 2.7 MB | 1.6 MB | | |
| RAM simulation | 20.2 s | 27.2 s | 19.8 s | 15.4 s | 15.3 s |
| | 97.1 MB | 97.6 MB | 97.1 MB | | |
| Video streaming | 27.5 s | 37.3 s | 8.5 s | 23.2 s | 3.3 s |
| | 180.2 MB | 184.6 MB | 7.4 MB | | |
| Face detection | 52.0 s | 70.1 s | 15.5 s | 47.6 s | 3.7 s |
| | 363.1 MB | 365.0 MB | 10.0 MB | | |
| KVM total migration time | Two layers | Three layers (app not found) | Three layers (app found) | KVM service downtime | |
| KVM total data transferred | | | | Two layers | Three layers |
| No application | 81.5 s | 141.8 s | 79.7 s | 55.8 s | 56.1 s |
| | 65.3 MB | 65.9 MB | 65.2 MB | | |
| Game server | 84.7 s | 142.0 s | 80.9 s | 60.2 s | 58.7 s |
| | 71.5 MB | 72.3 MB | 69.7 MB | | |
| RAM simulation | 95.9 s | 152.2 s | 93.0 s | 72.5 s | 72.0 s |
| | 170.0 MB | 178.4 MB | 170.4 MB | | |
| Video streaming | 120.9 s | 189.7 s | 86.4 s | 93.9 s | 61.3 s |
| | 251.3 MB | 270.5 MB | 87.3 MB | | |
| Face detection | 381.2 s | 558.3 s | 152.3 s | 355.0 s | 107.5 s |
| | 1,027.3 MB | 1,033.7 MB | 99.0 MB | | |

**TABLE 1**. Migration results for two-layer and three-layer configurations.

host system, and can therefore be more susceptible to attack from a compromised host.

## Two Layers vs. Three Layers

From the experimentation results, we see that most applications respond positively to the three-layer model. For LXC, when the application has been found at the migration target, Table 1 and Fig. 4 show that the Video Streaming and Face Detection applications have significant reductions in the total migration time compared to using only two layers, in the order of 3x for both. Even more significant is the reduction in service downtime for these applications, in the order of 7x and 13x, respectively. This service downtime reduction is particularly important for a seamless experience for the end user. The most significant is the reductions in the amount of data transferred over the network for migration, which are respectively in the order of 24x and 36x. This is particularly important for bandwidth or time constrained connections. Total migration times and data transfers for the Game Server, RAM Simulation, and No Application do not improve significantly, but are also not worse than with two layers when the application is found at the migration target. The relative reductions in total migration time and data transfers are smaller for KVM, because the application and associated data account for a smaller proportion of the total size of a KVM virtual machine.

We note that Video Streaming and Face Detection both have large filesystems compared to RAM Simulation and Game Server, and it is from this that the three-layer model derives its benefit. The larger the application or associated data, the greater the benefit from introducing an intermediate layer. This can be seen in Fig. 4, which shows how the migration of application data can account for a significant portion of the total time it takes to migrate an application.

The three-layer model does have a downside, in that if the application has not been found at the migration destination, the total migration time can be longer than with only two layers, because additional time is required for cloning the application into an instance and rsyncing the third layer (Fig. 4). However, the service downtime (which is usually the more critical factor for MEC applications) remains the same no matter whether the application is found at the destination or not, because the service is suspended only after the application
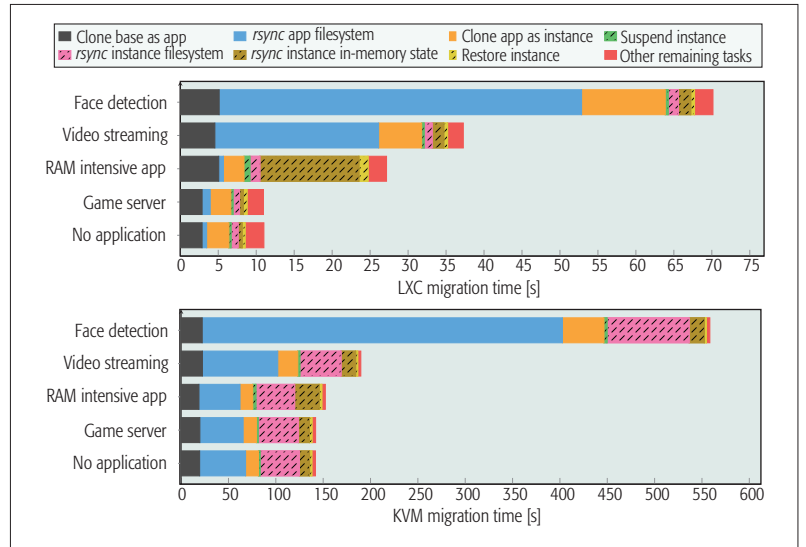


**FIGURE 4**. Application migration time broken down by stages, for the three-layer model when the application has not been found at the destination. The hatched stages represent those that contribute towards service downtime.

layer (containing an idle application) has been transferred.

Therefore, it is apparent that the benefits of the three-layer model need to be balanced with any trade-offs incurred. Factors such as how frequently the application is used and how much service downtime impacts the user experience should be taken into account when deciding whether to use two or three layers. Such decisions have to be made based on application characteristics. For example, it may be better for applications with small installation sizes and intensive RAM utilization to use only two layers for quicker migration.

A practical implementation of an MEC system can support both two-layer and three-layer models, since their underlying mechanisms are similar. The system can perform some simple profiling of applications and historical migration performance, and it can decide whether to use two or three layers on a case-by-case basis in real time.

## Open Issues

Our current implementation of containers uses the default directory backing store, and as a result, each additional layer duplicates the entire
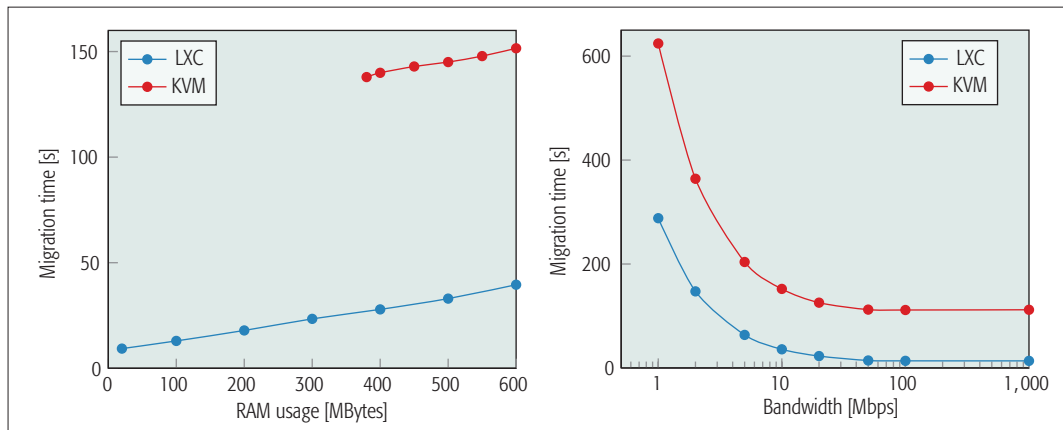


**FIGURE 5**. Total migration time under different RAM usage and bandwidth, for RAM simulation application.

Extensive experimentation results on the performance of different approaches for various applications under different scenarios have been presented. In general, the three-layer model with a container-based encapsulation environment gives the best overall performance, but other options may be preferred in specific cases.

filesystem of the previous layer plus the new data. For a layered setup like the one we demonstrated, a more ideal solution would be to try using an overlay filesystem such as overlayFS, which allows the sharing of lower-layer files with different upper layers. This would provide a much more efficient use of storage, and is much closer to the abstract model. However, LXC currently does not yet support overlayFS (as of version 1.1.5).

Another way of potentially improving performance, especially service downtime, would be to investigate the use of iterative migration. Iterative migration is the process of transferring memory pages while the service is running, so that when the service is finally suspended and migrated, in theory, only a small portion of the remaining in-memory state needs to be transferred.

For our experiments, LXC offered the greatest ease for running different applications without specialist knowledge in order to set them up. Other container technologies exist, among them Docker (www.docker.com) which has become the de facto standard for containerization. Future research should investigate whether Docker or other container technologies offer a better overall solution. To answer this question, consideration should be given as to what technology is likely to receive widespread adoption from industry, and therefore have the greatest impact. An interesting development to this is the work being carried out by the Open Container Initiative (www.opencontainers.org), which aims to create an open industry standard around container formats and run-times. Extensive industrial backing and a foundation based on Docker's format and run-times makes this a group to follow.

Besides migrating back-end applications between different MEC servers, another interesting aspect that is worth studying is the "vertical" migration of code and data between the mobile device and MEC, to strike a balance among device resource consumption and service quality. This is also known as application/computation offloading [14]. The challenge here is that the mobile device's operating system is often different from the server's operating system. A proper encapsulation mechanism that works on both platforms is needed to facilitate the migration. Since the layered framework presented in this article is applicable to a general class of encapsulation methods that supports the suspension of applications, we envision that a similar approach can be applied to live migration between mobile devices and MEC. A detailed study on vertical migration can be conducted in the future.

Future work should also study live migration performance under large-scale networked MEC systems. The performance of migration decision making in large-scale MEC environments has been mainly studied using simulations, where no real application migration is carried out [7, 10]. We have focused on the other end of the problem in this article, namely the implementation of live migration itself. In the future, it is worthwhile to study the migration of real applications in a realistic, large-scale networked system, which would connect the theoretical results in [7, 10] with the systems work in this article.

## Summary

We have presented a layered framework for service migration in MECs. The framework supports both container and VM technologies, and it can be easily implemented using existing functionalities of popular container and VM implementations. Extensive experimentation results on the performance of different approaches for various applications under different scenarios have been presented. In general, the three-layer model with a container-based encapsulation environment gives the best overall performance, but other options may be preferred in specific cases, as discussed. The three-layer model also allows the pre-caching of popular applications at MECs, so that the time required for future instantiation of such applications can be shortened. In addition, as migration is performed on the entire container or VM, the underlying service applications do not need to be specifically modified to support migration. This makes it easy to run existing applications in the migration framework. A future implementation may also specify an optional interface between the framework and the application, so that the application can announce when it prefers or does not prefer to be migrated, thereby improving migration performance.

## Acknowledgment

## References

[1] H. T. Dinh et al., "A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches," *Wireless Commun. and Mobile Computing*, vol. 13, no. 18, 2013, pp. 1587–1611.
[2] M. Satyanarayanan et al., "An Open Ecosystem for Mobile-Cloud Convergence," *IEEE Commun. Mag.*, vol. 53, no. 3, Mar. 2015, pp. 63–70.
[3] K. Ha et al., "Towards Wearable Cognitive Assistance," *Proc. ACM MobiSys*, 2014.
[4] X. Hou et al., "Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures," *IEEE Trans. Vehic. Technol.*, vol. 65, no. 6, June 2016, pp. 3860–73.
[5] "Mobile-Edge Computing — Introductory Technical White Paper," Sept. 2014; available: http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing.
[6] T. Taleb and A. Ksentini, "Follow Me Cloud: Interworking Federated Clouds and Distributed Mobile Networks," *IEEE Network*, vol. 27, no. 5, Sept. 2013, pp. 12–19.

[7] S. Wang, "Dynamic Service Placement in Mobile Micro-Clouds," Ph.D. dissertation, Imperial College London, 2015.

[8] Z. Becvar, J. Plachy, and P. Mach, "Path Selection Using Handover in Mobile Networks with Cloud-Enabled Small Cells," *Proc. IEEE PIMRC 2014*, Sept. 2014.

[9] K. Ha *et al.*, "Adaptive VM Handoff Across Cloudlets," Technical Report CMU-CS-15-113, CMU School of Computer Science, Tech. Rep., 2015.

[10] A. Ceselli, M. Premoli, and S. Secci, "Cloudlet Network Design Optimization," *Proc. IEEE IFIP Networking Conf.*, 2015, pp. 1–9.

[11] V. Medina and J. M. García, "A Survey of Migration Mechanisms of Virtual Machines," *ACM Comput. Surv.*, vol. 46, no. 3, Jan. 2014, pp. 30:1–33.

[12] T. Andersen, Live Migration of Linux Containers, Sept. 2014; available: https://tycho.ws/blog/2014/09/container-migration.html.

[13] J. Ahrenholtz, "Comparison of CORE Network Emulation Platforms," *Proc. IEEE MILCOM Conf.*, 2010, pp. 864–69.

[14] W. Zhang *et al.*, "Energy-Optimal Mobile Cloud Computing Under Stochastic Wireless Channel," *IEEE Trans. Wireless Commun.*, vol. 12, no. 9, Sept. 2013, pp. 4569–81.

[15] A. Machen *et al.*, "Poster: Migrating Running Applications Across Mobile Edge Clouds," *Proc. ACM MobiCom 2016*, Oct. 2016.

## BIOGRAPHIES

ANDREW MACHEN received the M.Sc. degree from Imperial College London, United Kingdom, in 2015. Before that, he received the M.Sc. and M.Eng. degrees from the University of Bristol, United Kingdom, respectively in 2009 and 2008. In the winter of 2015 and spring of 2016, he worked as an emerging technology intern at IBM, United Kingdom. His research interests include mobile cloud computing and application migration.

SHIQIANG WANG [S'13, M'15] received the Ph.D. degree from Imperial College London in 2015, and the M.Eng. and B.Eng. degrees from Northeastern University, China, in 2011 and 2009, respectively. He joined IBM T. J. Watson Research Center in 2016 as a research staff member. His current research focuses on theoretical and practical aspects of mobile edge computing, cloud computing, and machine learning. He has over 30 scholarly publications. He serves as an associate editor for *IEEE Access*, and has served as a technical program committee (TPC) member or reviewer for a number of international conferences and journals. He received the 2015 Best Student Paper Award of the Network and Information Sciences International Technology Alliance (ITA).

KIN K. LEUNG [F'01] received his Ph.D. degree from the University of California, Los Angeles in 1985. He joined AT&T Bell Labs in New Jersey in 1986 and worked at its successors, AT&T Labs and Lucent Technologies Bell Labs. Since 2004, he has been the Tanaka Chair Professor at Imperial College London. His current research focuses on protocols, optimization, cross-layer designs and multi-antennas in wireless networks. He received the Distinguished Member of Technical Staff Award from AT&T Bell Labs (1994) and the Lanchester Prize Honorable Mention Award (1997). He was elected an IEEE Fellow (2001), received the Royal Society Wolfson Research Merits Award (2004–09) and became a member of Academia Europaea (2012). He was a member (2009–11) and the chairman (2012–15) of the IEEE Fellow Evaluation Committee for the Communications Society. He has served as an editor for the IEEE JSAC: Wireless Series, *IEEE Transactions on Wireless Communications*, *IEEE Transactions on Communications*, the *ACM Computing Survey* and the *International Journal on Sensor Networks*.

BONG JUN KO is a research staff member, research manager at IBM T. J. Watson Research Center, where he is currently leading a research effort to develop technologies that enable data analytics and machine learning for Internet of Things (IoT) applications. He has also worked on research projects related to distributed network management, micro cloud, and network analytics. Prior to joining IBM Research, he worked as a senior member of research staff at Philips Research North America, and as a research engineer at LG Electronics, Korea. He received a Ph.D. degree in electrical engineering from Columbia University in 2006, and B.S. and M.S. degrees from Seoul National University in South Korea. He received the Best Paper Awards at IEEE ICNP 2003 and IEEE/IFIP IM 2013, and multiple corporate awards at IBM such as IBM Master Inventor (2013, 2016) and Outstanding Technical Achievement Award (2015).

THEODOROS SALONIDIS [S'98, M'04] received the diploma in electronic and computer engineering from the Technical University of Crete, Chania, Greece, in 1997, and the M.S. and Ph.D. degrees in electrical engineering from the University of Maryland, College Park, in 1999 and 2004, respectively. He is a research staff member with the IBM T. J. Watson Research Center, Yorktown Heights, NY. Prior to IBM, he held research positions at Thomson/Technicolor (2007-2012), Intel Research (2006) and Rice University (2004-2006). His current research interests are in the areas of mobile networks, cloud computing systems and distributed machine learning. He served as technical co-chair of IEEE SECON 2012, and served on the program committee for numerous networking conferences including IEEE ICNP, IEEE Infocom, ACM CoNext and ACM MobiHoc. He received the Best Student Paper Award from ACM Sigmetrics 2015 and the Best Demo Award from ACM MobiCom 2009.