Virtual Machine Migration for IoT Applications

Yutao Tang
Sam's Club Lab
Yutao.Tang@walmart.com

Zhengrui Qin
Northwest Missouri State University
zqin@nwmissouri.edu

Shanhe Yi
VMware
yshanhe@vmware.com

Qun Li
College of William and Mary
liqun@cs.wm.edu

ABSTRACT

To integrate heterogeneous devices in IoT (Internet of Things), we design and implement an IoT system consisting of smart meters, smartphones, and cloud servers. In our system, a smartphone can rent its idle resources to the third party, which can install an application to conduct various tasks, such as collecting and processing data from nearby smart meters. Built on the top of Xen hypervisor and MiniOS, our system cannot only ensure smartphone's security by providing isolated resources to the application, but also guarantee data integrity and privacy of the application. We are the first to build a lightweight virtual machine migration system on an ARM-architecture smartphone.

CCS CONCEPTS

• **Security and privacy** → *Security services*;

KEYWORDS

Virtual machine migration, IoT, Mobile computing

ACM Reference Format:
Yutao Tang, Zhengrui Qin, Shanhe Yi, and Qun Li. 2019. Virtual Machine Migration for IoT Applications. In *The Fourth ACM/IEEE Symposium on Edge Computing (SEC 2019), November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3318216.3363383

1 INTRODUCTION

Recently, Internet of Things (IoT) has emerged as the future of Internet. However, one big issue of the current IoT is that those devices are mostly resource-limited in collecting, on-site processing and sharing large volume of data for advanced analytics. One way to mitigate this problem is to back IoT systems with cloud services, but it cannot catch up with the pace of the increasing demands for cost efficiency, low latency, and mobility support. As a result, edge computing is recently proposed to push computations from cloud to the edge of networks. While there is not restriction on what type of devices can be the edge node, most existing literature or proposals prefer resource-rich edge nodes, like servers, high-end desktops or laptops. However, deploying those devices usually means extra cost and low mobility, which are all important inhibitor factors considered by end users.

Therefore, in this paper, we explore the feasibility and techniques that can turn smartphones into lightweight edge nodes to enhance current IoT systems. We argue that recruiting smartphones as edge nodes can be a complementary edge computing deployment method. For example, there is already a trend of merging IoT and smartphones in Android Things where IoT device will have similar architecture as smartphone [7].

Our motivation is that computing resources of smartphone are not fully utilized all the time. Hence, it is possible to use an incentive mechanism to recruit smartphones as edge nodes to complement or substitute the fixed sink nodes in current IoT systems. Different from fixed sink nodes, smartphones can collect data from smart meters along the paths of smartphone owners, enhancing the mobility of the IoT network. They can process data at a very early stage and help scale up IoT easily at low cost. An illustrative example would be a smart city with massive smart meters deployed at various locations. As mobile users come and go, their smartphones can collect data from smart meters within the communication range. At the same time, smartphones can help process the collected data, store it, and send it to backend server or cloud services.

In general, our proposed system consists of four entities: *application administrator*, *cloud service*, *smartphones*, and *meter*, as shown in Fig. 1. The application administrator issues an application request to the cloud service, which relays the request to a set of chosen smartphones. Once receiving the

SEC 2019, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363383

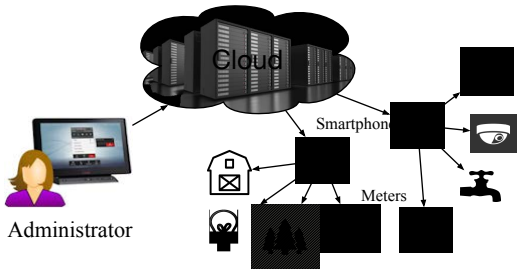

Figure 1: The scenario where the smartphone serves as the hub of IoT networks.

request, each smartphone will allocate resources to run the application. The application can interact with the nearby meters through wireless links, such as collecting data from meters, processing data, and sending feedback or control command back to meters. The application can also sends pre-processed data back to the application administrator through the cloud service.

There are two major challenges to realize the above system. The first challenge is the continuity of application execution. A smartphone may abort the execution of the application, due to either the smart phone is out of the communication range of the meters or its available resources run out. In these cases, the smartphone must migrate the unfinished application to another smartphone nearby, or hand it over to a normal sink node, or upload it to cloud services for further processing. The second challenge is the security. From either the perspective of the smartphone or that of the IoT application, the other entity is untrusted. On one hand, although the owner of the smartphone is willing to sell her idle resources, she must be guaranteed that there is no security and privacy risk or hardware/software damage to her own device. On the other hand, the IoT system must be assured that the smartphone keeps the integrity of its data, processes the data as it is instructed to, and does not leak any content of the application.

In addressing the challenges, we propose to construct virtual machine on smartphones with migration scheme. We build an Xen hypervisor above the smartphone hardware. On top of the hypervisor, there is a special virtual machine, called Dom0, which is trusted. Then we create multiple MiniOS, each of which is supervised by Dom0 and runs an application solely. In this way, each MiniOS is an isolated resource block and does not affect other resources of the smartphone. At the same time, the user of the smartphone cannot access the data that is collected and processed in the MiniOS. MiniOS occupies only a small portion of resources, which is suitable to run on smartphone hardware. As the creation, deletion, and migration of MiniOS can be done in real time, the continuum of the IoT application is guaranteed and the resources on smartphone can be efficiently utilized.

In summary, our main contributions are as follows:

- We have proposed an IoT system that harnesses idle resources of untrusted smartphones to run applications from a third party.
- We have designed a virtualization based isolation with Xen and MiniOS on top of ARM architecture.
- We have designed an efficient virtual machine migration scheme.
- We have implemented a prototype system.
- We are the first to migrate MiniOS virtual machine on smartphone of ARM architecture.

2 RELATED WORK

In this section, we will briefly review the role of smartphone in IoT application and and point out the difference between our work and the existing work. As the focus of our work is on application migration on smartphones, we will also review work along this line.

2.1 Smartphone in IoT

Smartphones have been playing important roles in IoT applications. For instance, they have been utilized for data collection and data relay in wireless networks, such as work [12]. Furthermore, they have been harnessed in IoT-related environments in work [13]. Our system shares distinct difference as the smartphones in our system are not trusted. Due to the mobility and serving purpose, smartphones are allowed to join or leave the IoT system at will. Therefore, with our system, IoT devices are not necessarily bounded to a single smartphone that has to be owned by the same person. Most importantly, our system solves the problem of continuum execution by migrating the unfinished application seamlessly to other smartphones.

2.2 Virtualization and Migration

Researchers have built systems based Xen hypervisor [4] and MiniOS, such as ClickOS [10] and MirageOS [8]. While ClickOS is built on x86 architecture and MirageOS is able to be built on both x86 and ARM architectures, they do not consider migration. Work [9], being along the line of using minimalistic VMs, makes use of MiniOS. However, they focus more on consolidating more VMs on high-volume server. Rump kernel [3] is a unikernel that allows applications to be linked into a standalone executable running on the Xen hypervisor without an operating system. It shares some similarities with our design in this paper, but faces different challenges as our system is on smartphones with ARM architecture.

As to migration, most of existing work focuses on full operating system migration on x86 [5, 6, 11], either on commodity computers or on servers. We are the first to migrate the lightweight MiniOS on an untrusted smartphone.

3 SYSTEM OVERVIEW

In this section, we will present the assumption, design goals and system architectures.

3.1 Trust Model

In our system, we focus on the security concern of the smartphone and the IoT application, rather than other components or any communication link. Thus, we assume that the cloud service is trusted, which is usually the case when using large commercial cloud services like Amazon EC2, Microsoft Azure, and Google Cloud Platform. We further trust the meters themselves, since they are controlled and configured by the application administrator. However, from the perspective of the application administrator, the smartphones are not trusted. From the perspective of the mobile users, the meters and the application are not trusted.

We assume that the smartphones support virtualization. We also assume that the Xen hypervisor is trusted. Furthermore, we assume that the the initial domain started by Xen on boot, i.e. *Dom0*, is trusted.

3.2 Design Goals

We are to build a system that fulfills the following design goals:

- **G1** Isolated resources. The dedicated resources for rent should be isolated from other resources on the smartphone.
- **G2** Correct computation. All computations must be conducted correctly as the application instructs.
- **G3** Data privacy. The smartphone must have no access to any data run by the application, neither does the application have access to smartphone's data.
- **G4** Data integrity.The smartphone is forbidden to manipulate any data passing through it by the IoT application.
- **G5** Flexibility. The system must support fast allocation and revocation of resources.

3.3 Architecture on Smartphone

As mentioned in Section 1, our system has four components: application administrator, cloud service, smartphone, and meters. The core is the smartphone, while others are the same as those in a standard IoT system. Hence, our focus here is on the architecture design on the smartphone, where the migration takes place.

The smartphone provides a virtual machine architecture consisting of two major components: Xen hypervisor and MiniOS. As shown in Fig. 2, Xen hypervisor runs on the top of the smartphone's hardware. On top of Xen hypervisor, there are multiple virtual machines, including Dom0, Dom1 and multiple MiniOS instances. As the initial domain started by Xen hypervisor, Dom0 is trusted and can access the hardware directly. Dom1 is the Android operating system for the smartphone owner. Each MiniOS is started for an IoT application exclusively, and it is configured and controlled by Dom0.

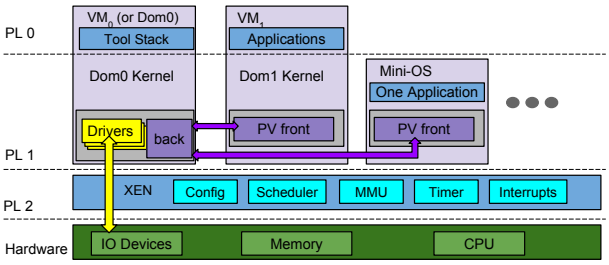

Figure 2: The architecture on the smartphone.

3.4 System Procedure

Suppose the application administrator has an IoT application to run. Our system takes the following procedures (please refer to Fig. 1):

(1) The administrator creates the MiniOS image of the application and sends it to the cloud service (e.g. an image registry service);
(2) The cloud service store and relays the MiniOS image to Dom0 of any available smartphone;
(3) AT the smartphone side, the Dom0 creates a MiniOS instance with the given image;
(4) The created MiniOS starts the application and execute its tasks;
(5) If MiniOS has finished all the tasks of the application, Dom0 revokes all resources and sends the result of the application back to the cloud service;
(6) If the smartphone is not able to provide resources while the task of the application is not finished yet, MiniOS creates the migration image of the unfinished tasks of the application, and Dom0 revokes all resources and sends the migration image back to the cloud service;
(7) The cloud service sends the result to the administrator if the application is finished, or finds another available smartphone to continue the unfinished application by sending it the original MiniOS image and the migration image.

The most challenging part of this system is the migration procedure, on which we focus in this paper. We will detail the migration procedure in Section 4.

4 MIGRATION PROCEDURE

The most important advantage of our system is the flexibility of smartphone participation, which allows smartphones join and leave the IoT system at any moment. It is inevitable

that a smartphone may leave the system before the finish of tasks of an application. As a result, the unfinished application must be picked up by another smartphone seamlessly. In the following, we will detail how to migrate the unfinished application from one smartphone to another.

4.1 What to Migrate

To migrate the unfinished application as quickly as possible, we first must determine the only necessary data that needs to be migrated. The data can be categorized to memory data, CPU states, and disk data. Since the disk migration is relatively easy, just by copying all data in source disk to the destination disk, and CPU state will be stored into the memory when the migration process runs, our problem boils down to migrate the memory data.

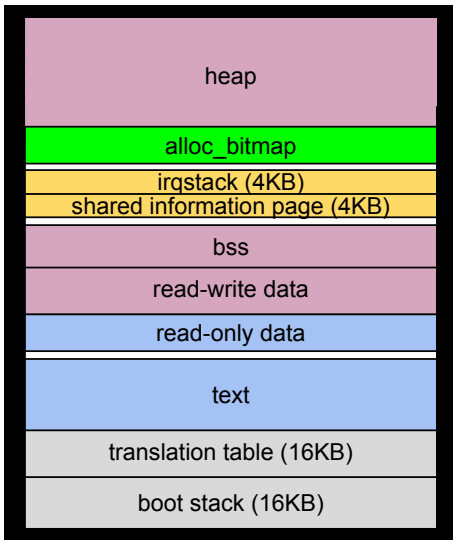

Figure 3: Virtual address space.

Next we need to analyze the memory usage in MiniOS. The virtual memory address space is shown Fig. 3, and we can classify the memory into three categories:

(1) The memory that is not modified during system running such as the *read-only data* and the *text* segment, and that is used by by system processes and system structures to support system running such as the *boot stack* segment and the *translation table* segment;
(2) The memory that is used by drivers or system processes for communication between MiniOS and Xen, such as the *shared information page* segment.
(3) The memory that is used by application processes, including all application data, some system data in the *read-write data* and the *bss* segment, and all memories allocated by *malloc* function by application processes.

We divides the MiniOS virtual machine into two parts: *Base* and *Delta*. Base contains resources that do not change, and Delta contains resources that change as the application is running. We take all resources allocated before the creation of application processes as Base, and all resources allocated thereafter as Delta.

It is clear that the memory of category 1 belongs to Base and that of category 3 belongs to Delta. However, it is not easy to infer which the memory of category 2 belongs to. To solve this uncertainty, we design to shut off all device drivers before migration and initialize them in the destination smartphone before resuming the application. And we require the application processes access drivers only by APIs provided by our system. In this way, the memory of category 2 does not need to be migrated, since it is freed before migration. Thus, our work boils further down to migrate the memory of category 3.

4.2 How to Migrate

Once we know the concept of Delta for migration, the next is how to migrate it. The most important challenges are:

4.2.1 How to design the migration process. We need a dedicated process to handle the migration. Since this process runs as a standard process in MiniOS, it also consumes memory resource, which could be mingled together with the memory chunks which should be migrated.

4.2.2 Where to save Delta. As we have mentioned before, we shut off all devices used by the application processes, including networking. Therefore, we cannot transmit Delta to the cloud service during migration. We cannot save Delta into memory either, since the smartphone may have limited memory resource. As a result, we need a storage space used exclusively by the migration process.

4.2.3 How to locate Delta. The memory resource owned by application processes are located in different memory address spaces. We design a scheme to locate the data indispensable for migration and we explain it in the following steps:

1) **Allocate Memory Early**. We design a system process called *migration* to handle the MiniOS migration. MiniOS utilizes *migration* and three other processes, *xenstore*, *shutdown* and *idle*, to support system running. Before running application processes, we create all system processes and pre-allocate enough memory to *migration* process such that the system processes have the same memory distribution for every boot and *migration* process does not need to allocate memory later. Second, if we plan to obtain a memory snapshot, we do so only when *xenstore* has empty buffers to make sure no new heap memory allocated.

2) **Save Data to Disk**. We design to store Delta to disk. We create an extra para-virtualized disk for MiniOS, called *Disk0*, to exclusively save the migration data.

3) **Mine Necessary Data for Migration**. To obtain the modified memory, we keep two snapshots of the selected memory at different time. The first snapshot, called *snapshot0*, is obtained at the point right before the application processes are created, and the second, called *snapshot1*, is obtained at the time when the migration begins. The *migration* process performs XOR operation on the two snapshots, and save the XORed results into *Disk0*.

4) **Use CPU Exclusively**. When the *migration* process performs MiniOS migration, it has to keep the memory distribution unchanged. Therefore, we design to stop all other processes from executing, otherwise their execution can modify the memory distribution.

4.3 The Migration Flow

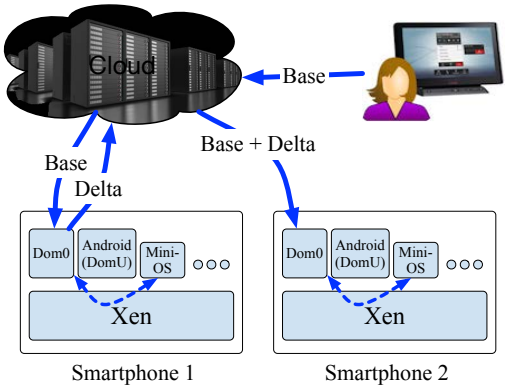

Figure 4: The migration procedure.

Suppose smartphone 1 stops providing resources due to whatever reasons and wants to migrate the unfinished application to smartphone 2, the corresponding migration flow is illustrated in Fig. 4:

(1) Smartphone 1 generates Delta and stores it into *Disk0*.
(2) Dom0 of smartphone 1 transmits Delta to the cloud service.
(3) The cloud service sends Delta and the associated Base to Dom0 of smartphone 2.
(4) Dom0 creates a MiniOS with Base and Delta.
(5) MiniOS resumes the application.

5 IMPLEMENTATION

To validate our system design, we have implemented a prototype system on top of Arndale Board-K development board, which is same as the boards used in Nexus 10 and Chromebook. This board has two advantages for implementing our prototype system. One is that it has an ARM Cortex-A15 processor, which provides virtualization extension and is officially supported by Xen hypervisor. The other is that Linaro [1] provides open-sourced bootloader of this board.

5.1 Device APIs

We implement our system with the following suports:

Network. Network is required for communication with the cloud service, such as uploading Delta, downloading Base and/or Delta. In our system, we choose lwIP (lightweight IP) as the TCP/IP stack. lwIP is an open-source project designed for embedded system [2]. Occupying only a very small memory footprint (512KB), lwIP can provide applications the basic network access functionalities.

Disk. We provide APIs for the application to access disk, instead of providing a file system, because there is only one application in MiniOS. Our system needs two disks, one for the application itself (*Disk1*), the other for the migration data (*Disk0*).

GPS. In our system, GPS is used to measure the distance between the smartphone and the meters and thus to trigger the migration when the smartphone is out of the communication range of the meters. The Arndale development board, however, does not have a GPS module. We hence design a GPS emulator to generate GPS data, which consists of three components: a generator, a backend driver, and a frontend driver.

Bluetooth. Bluetooth is used by the application to communicate with the meters. As the Arndale development board does not have a bluetooth module, we emulate the bluetooth device, similar as GPS.

5.2 Application Programming

The application running in MiniOS is required to be compiled into a user library and uses *main_app* as the entry point. It can only utilize APIs listed above to access devices, since it is forbidden to modify the system memory. Furthermore, it must provide a network reconnection function in case that a new IP address is assigned to its MiniOS after being migrated to a new smartphone.

5.3 Migration Implementation

When MiniOS starts to boot, it first initializes all system resources. Then, as shown in Fig. 5, MiniOS creates four system processes: *xenstore*, *shutdown*, *idle*, and *migration*. *migration* is created earlier than any of the application processes and serves as the entry point of the application code. Furthermore, it calls the *malloc* function only once to allocate memory large enough to finish all operations.

6 EVALUATION

In this section, we will evaluate save/resume time of our prototype system.

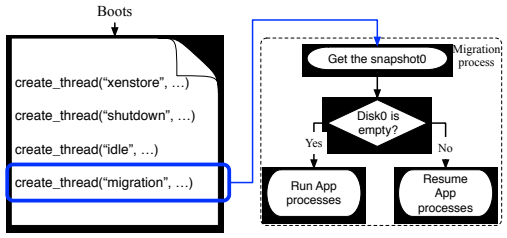

Figure 5: The migration implementation.

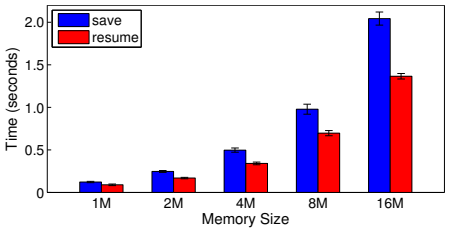

Figure 6: The save time and the resume time for different memory size.

6.1 Environment Setup

The smartphone board in our prototype system is the Arndale Board-K development board. The board has the Exynos 5 Dual SoC, 1 GB RAM, and a 64 GB external SD card. It is also equipped with a 1.7 GHz dual-core ARM Cortex-A15 processor that provides hardware virtualization support. The cloud is emulated by a Lenovo Y480, equipped with a 2.4G Hz I7 processor, 8G memory, and a Qualcomm Atheros AR8161 gigabit Ethernet. The development board and the cloud are wired through a router. We use Xen4.4.1 for the hypervisor, u-boot from Linaro for the bootloader, and Linux 3.19.7 for Dom0 and Dom1. The application, i.e., MiniOS image, is provided to Dom0 by the cloud.

6.2 Save/Resume Time

The most significant advantage of our system is the migration of the (unfinished) application from one smartphone to another whenever necessary. Here we investigate how fast the migration can be conducted. We focus on the *save time* and the *resume time* while ignoring the networking time from a smartphone to the cloud and that from the cloud to another smartphone. The save time is the time span from the time when the migration is triggered to the time once all the migration data, i.e. Delta, is stored into the disk (*Disk0* and *Disk1*). The resume time is the time span from the time when Dom0 of Xen receives all the migration data (i.e. Delta+Base) to the time when the (unfinished) application is ready to continue running. In our experiment, we choose a matrix multiplication task as the application, and deliberately choose different matrix size such that different size of memory has to be allocated for the migration data (Delta). Fig. 6 shows the average of the save/resume time of 10 round tests along with error bar for different memory size. We can see that the save time is a little longer than the resume time, and both scale out almost linearly with regard to the memory size.

7 CONCLUSION

In this paper, we have proposed an Edge-IoT system that consists of smartphones, meters, the cloud service, and the application administrator. The smartphones are not trusted, and they can join and leave the system at will. We implemented a prototype system on the smartphone side. We have built a Xen hypervisor on the Arndale development board, and created multiple MiniOSes on top of Xen hypervisor. We are the first to migrate MiniOS virtual machine on ARM architecture. Even though the application and the smartphone are mutually untrusted, they are guaranteed the protection of data integrity and data privacy. We have also carefully designed the migration process such that only the necessary data is migrated.

REFERENCES

[1] 2017. Linaro. https://www.linaro.org. (2017).
[2] 2017. MiniOS on cubieboard2. http://savannah.nongnu.org/projects/lwip/. (2017).
[3] 2017. Rump kernel. http://rumpkernel.org. (2017).
[4] Paul Barham and others. 2003. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.
[5] David Breitgand, Gilad Kutiel, and Danny Raz. 2010. Cost-aware live migration of services in the cloud.. In *SYSTOR*.
[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.
[7] Google Inc. 2017. Android Thing. https://developer.android.com/things/index.html. (2017).
[8] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, and others. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *NSDI*. 559–573.
[9] Filipe Manco, Joao Martins, Kenichi Yasukata, Jose Mendes, Simon Kuenzer, and Felipe Huici. 2015. The Case for the Superfluid Cloud.. In *HotCloud*.
[10] Joao Martins *et al.* ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, 459–473.
[11] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *Pervasive Computing* 8, 4 (2009), 14–23.
[12] Shusen Yang and others. 2013. Selfish mules: Social profit maximization in sparse sensornets using rationally-selfish human relays. *Selected Areas in Communications, IEEE Journal on* 31, 6 (2013), 1124–1134.
[13] Thomas Zachariah and others. 2015. The Internet of Things Has a Gateway Problem. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 27–32.