Research on Efficiency and Security for Emerging Distributed Applications

Zijiang Hao
William & Mary - Arts & Sciences, zhao01@email.wm.edu

Research on Efficiency and Security for Emerging Distributed Applications

Zijiang Hao

Tianjin, China

Master of Engineer, Tsinghua University, 2007
Bachelor of Science, Tsinghua University, 2004

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May, 2019

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Zijiang Hao

Zijiang Hao

Reviewed by the Committee, May 2019

[illegible]

Committee Chair
Professor Qun Li, Computer Science
College of William & Mary

[illegible]

Associate Professor Peter Kemper, Computer Science
College of William & Mary

[illegible]

Associate Professor Gang Zhou, Computer Science
College of William & Mary

[illegible]

Assistant Professor Zhenming Liu, Computer Science
College of William & Mary

[illegible]

Associate Professor Gexin Yu, Mathematics
College of William & Mary

ABSTRACT

Distributed computing has never stopped its advancement since the early years of computer systems. In recent years, edge computing has emerged as an extension of cloud computing. The main idea of edge computing is to provide hardware resources in proximity to the end devices, thereby offering low network latency and high network bandwidth. However, as an emerging distributed computing paradigm, edge computing currently lacks effective system support. To this end, this dissertation studies the ways of building system support for edge computing.

We first study how to support the existing, non-edge-computing applications in edge computing environments. This research leads to the design of a platform called SMOC that supports executing mobile applications on edge servers. We consider mobile applications in this project because there are a great number of mobile applications in the market and we believe that mobile-edge computing will become an important edge computing paradigm in the future. SMOC supports executing ARM-based mobile applications on x86 edge servers by establishing a running environment identical to that of the mobile device at the edge. It also exploits hardware virtualization on the mobile device to protect user input.

Next, we investigate how to facilitate the development of edge applications with system support. This study leads to the design of an edge computing framework called EdgeEngine, which consists of a middleware running on top of the edge computing infrastructure and a powerful, concise programming interface. Developers can implement edge applications with minimal programming effort through the programming interface, and the middleware automatically fulfills the routine tasks, such as data dispatching, task scheduling, lock management, etc., in a highly efficient way.

Finally, we envision that consensus will be an important building block for many edge applications, because we consider the consensus problem to be the most important fundamental problem in distributed computing while edge computing is an emerging distributed computing paradigm. Therefore, we investigate how to support the edge applications that rely on consensus, helping them achieve good performance. This study leads to the design of a novel, Paxos-based consensus protocol called Nomad, which rapidly orders the messages received by the edge. Nomad can quickly adapt to the workload changes across the edge computing system, and it incorporates a backend cloud to resolve the conflicts in a timely manner. By doing so, Nomad reduces the user-perceived latency as much as possible, outperforming the existing consensus protocols.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First, I thank my doctoral advisor Dr. Qun Li for his continuous support during my Ph.D. study and for his patient guidance and constructive advice that helped me obtain all my research outcomes. He taught me everything about doing good research and, more importantly, leading a fruitful life. It has been my greatest honor to have him as my advisor.

Next, I thank my dissertation committee members: Dr. Peter Kemper, Dr. Gang Zhou, Dr. Zhenming Liu, and Dr. Gexin Yu, for all of their support.

I thank all my co-authors: Yifan Zhang, Zhengrui Qin, Ed Novak, Nancy Carter, Yutao Tang, Shanhe Yi, Dr. Gang Zhou, Dr. Guoliang Xing, Xin Qi, David T. Nguyen, Ge Peng, Qing Yang, Dr. Weisong Shi, Quan Zhang, Qingyang Zhang, and Raymond Ji, for their insightful suggestions, valuable feedback, and tremendous effort.

I thank all professors who instructed me in their classes at the College of William and Mary: Dr. Virginia Torczon, Dr. Andreas Stathopoulos, Dr. Evgenia Smirni, Dr. Weizhen Mao, Dr. Haining Wang, Dr. Gang Zhou, Dr. Denys Poshyvanyk, and Dr. Thomas Dillig. Their exceptional instruction has influenced my study, research, and life in many ways.

I thank our department chair Dr. Robert M. Lewis, graduate director Dr. Pieter Peers, and the administration team, Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes for creating an excellent research environment.

I thank all members from our research group for providing useful insights: Fengyuan Xu, Wei Wei, Hao Han, Yifan Zhang, Zhengrui Qin, Ed Novak, Nancy Carter, Yutao Tang, Shanhe Yi, Cheng Li, Lele Ma, Qi Xia, and Zeyi Tao.

Finally, I thank my family for their endless support, encouragement, and love.

I would like to dedicate this dissertation to my parents, Changping Hao and Guihua Dai, who provided endless support and love throughout my time at the College of William and Mary.

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Since Amazon released its EC2 (Elastic Compute Cloud) [5] product in 2006, cloud computing has become increasingly important in people's daily life. By providing elastic hardware resources, including processing resources and storage resources, at the data centers residing at the core of the Internet, cloud computing enables a spectrum of applications that have profoundly impacted the contemporary computational patterns for both industrial and individual uses. Companies can benefit from cloud computing by executing large batch-oriented tasks on cloud servers, and individual users can rely on remote clouds to perform resource-intensive computations for their client devices. Because cloud computing has brought so many applications into reality, commercial cloud platforms, such as Amazon AWS [4], Microsoft Azure [65], and Google Cloud [38], have been successively put into operation in recent years.

Nevertheless, cloud computing suffers from a severe problem when serving end devices at the edge of the Internet. Since cloud data centers usually reside at the core of the Internet, it is always the case that end devices have a long network distance to the remote clouds, which leads to significant network delay perceived by the end users. This is unacceptable for many application scenarios, such as latency-sensitive mobile-cloud and IoT-cloud applications, where the end devices are mobile devices such as smartphones and IoT (Internet of Things) devices, respectively.

In light of this situation, edge computing [84] (also known as fog computing [102, 94] and cloudlets [85]) has been proposed as an extension of cloud computing. By providing hardware sources at the edge of the Internet, end devices can be served with much lower network latency, thereby greatly improving the user experience for latency-sensitive end-remote applications.

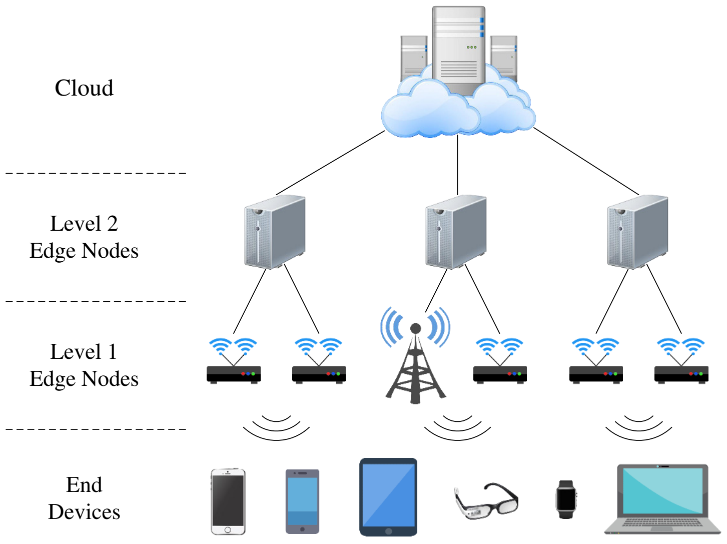


Figure 1.1: A typical edge computing architecture.

Figure 1.1 illustrates a typical architecture of edge computing. End devices are wirelessly connected to the Level 1 edge nodes, which are usually Wi-Fi access points and cellular towers. Behind the Level 1 edge nodes are the Level 2 edge nodes, which have a farther network distance to the end devices than the Level 1 edge nodes. There is also a backend cloud behind the Level 2 edge nodes that resides at the core of the Internet. The end devices will perceive lower latency when computations are performed at the Level 1 edge than at the Level 2 edge and the cloud. On the other hand, the Level 2 edge nodes usually possess more powerful hardware resources than the Level 1 edge nodes and can thus perform more computations simultaneously than the Level 1 edge nodes. Note that Figure 1.1 merely illustrates a possible architecture of edge computing. In other edge computing architectures, it is possible that there is no backend cloud, and/or there are only one level of edge nodes or more than two levels of edge nodes.

Despite its advantages, edge computing currently lacks system support that makes efficient use of the underlying hardware resources and provides powerful, fundamental services for edge applications. For this reason, this dissertation investigates how to build effective system support for edge computing in various application scenarios. We summarize as below three ways of building such system support that can benefit a vast majority of application scenarios in edge computing from our point of view.

- There are many applications utilized by a great number of users nowadays. These applications usually have evolved for many versions and are mature in their design and implementation. Nevertheless, edge computing is a new distributed computing concept, so these applications are not designed for edge computing and cannot be directly executed in edge computing environments. It is thus attractive to build system support for these existing applications, making them executable in edge computing environments without modification, so that the application users can conveniently enjoy the benefits brought by edge computing.
- Edge computing is a new concept and has many distinguishing characteristics, so developers need an efficient programming model of edge computing to reduce the programming effort they must put into implementing new edge applications. Therefore, designing an efficient programming model of edge computing and providing system support for it, thereby allowing developers to implement their edge applications with as little programming effort as possible while guaranteeing the efficiency and robustness of the applications, is also an important way to build system support for edge computing.
- Edge computing falls into the category of distributed computing, while the most important fundamental problem in distributed computing, from our understanding, is the consensus problem. Existing consensus solutions are not tailored for edge computing, so they cannot fully explore the potential of edge computing in achieving fast

consensus. Therefore, building system support for achieving fast consensus in edge computing environments, making the entire system agree on the same decision as quickly as possible, can benefit a large number of edge applications.

We believe that building system support for edge computing in the aforementioned three ways will benefit a vast majority of, if not all, application scenarios in edge computing. To this end, we have conducted three research projects to investigate how to build system support for edge computing in these three ways. Figure 1.2 demonstrates these research projects and their goals.

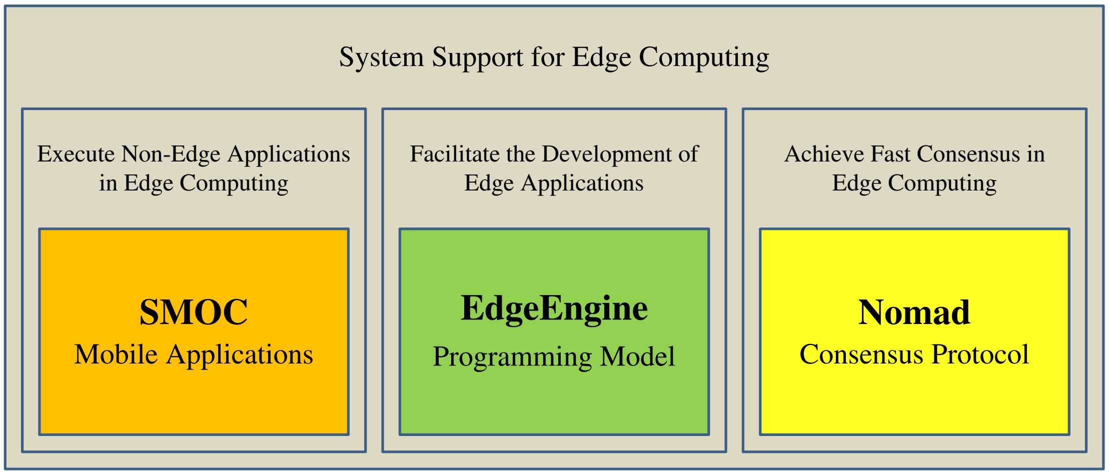

System Support for Edge Computing
Execute Non-Edge Applications in Edge Computing
Facilitate the Development of Edge Applications
Achieve Fast Consensus in Edge Computing
SMOC
Mobile Applications
EdgeEngine
Programming Model
Nomad
Consensus Protocol

Figure 1.2: The research projects discussed in the dissertation and their goals.

In the first project, we try to build a platform called SMOC to support executing off-the-shelf mobile applications in an edge computing environment. We focus on mobile applications in this project because doing so can benefit a great number of users, given that there are hundreds of billions of mobile applications in the market nowadays. In the second project, we try to design an edge computing framework called EdgeEngine that consists of a programming interface and a middleware running on top of the edge computing infrastructure. Developers can build their edge applications through the programming interface with minimal programming effort, while the middleware guarantees good efficiency and robustness of the applications. In the third project, we try to design a consensus protocol called Nomad that is tailored for edge computing. It achieves low user-perceived latency by quickly adapting to the workload changes across the system and utilizing a backend cloud to resolve the conflicts at the edge in a timely manner. In what follows, we will introduce these research projects with more details.

1.0.1 SMOC: Supporting Mobile Applications in Edge Environments

In this project, we aim to support the existing mobile applications in edge environments without any modification. It is known that most mobile applications are ARM-based [112], while edge servers are always x86 [113] or x64 [114] PC servers. Consequently, without proper system support, mobile applications cannot be executed on edge servers unless being ported to the x86 or x64 architecture. However, porting source code is a tedious, time-consuming, and error-prone activity [19, 80], and sometimes the source code is even unavailable. As such, we build a mobile-edge platform called SMOC to solve this problem. SMOC provides a running environment identical to that of the mobile device on the edge, so that mobile applications can be directly executed on the edge servers. Moreover, SMOC ensures secure computation for mobile applications by taking advantage of edge computing. The operating systems on mobile devices are complex, potentially containing a good number of flaws [20]. Therefore, if a mobile user wants to run some confidential applications on the mobile devices, such as an online-banking application, she must face the risk that the mobile device may leak her confidential information (e.g., her password input) to the attackers. SMOC protects the user input by utilizing the mobile device as an input/output terminal and exploiting hardware virtualization on the mobile device.

SMOC works in two modes, a normal mode and a secure mode. In the normal mode, the mobile user can run the applications on the mobile device as usual. In the secure mode, SMOC exploits the hardware virtualization feature on the mobile device. It runs a thin hypervisor layer on the mobile device, which resides between the hardware and the mobile operating system. The hypervisor is considered safe because it is so small that every line of its source code can be manually inspected. The hypervisor intercepts the user input from the hardware, bypassing the potentially compromised operating system, and sends the input to a remote virtual machine (VM) on the edge. The edge VM emulates the same executing environment as the mobile device. Any computation that requires a high secure level can be run in the VM. As user input is fed into the computation by the VM, it can be executed normally on the edge. The VM also forwards the output of the computation, including screen frames and sound, to the hypervisor, which will render the output on the local hardware. By this means, the user can run risky applications remotely, without the concern that the potentially compromised operating system will learn any confidential information from her input.

SMOC allows mobile users to switch the execution of applications between the secure mode and the normal mode. It does so by building a distributed file system spanning across the mobile device and the edge VM. Users can freely change the running location of their applications. When they prefer a higher security level, they may run their applications in the secure mode, and when they prefer lower latency, they may switch their applications to the normal mode. Our evaluation shows that SMOC is quite convenient to mobile users. Furthermore, SMOC introduces reasonable overhead when running in the secure mode, and negligible overhead when running in the normal mode.

1.0.2 EdgeEngine: Facilitating the Development of Edge Applications

In this project, we aim to build system support to facilitate the development of edge applications. We envision that building edge applications can be very complicated. On one hand, an edge application may involve many *hardware entities*, including one or more end devices, one or more edge servers, and zero or one remote cloud server. Coordinating all these hardware entities and making them function correctly in all cases, including potentially many corner cases, is not an easy job. Consequently, the implementation may be inefficient or even full of defects. On the other hand, developers must strive for not only the business logic of their applications, but also many fundamental services, such as data dispatching, data synchronization, task scheduling, lock management, etc. Implementing and debugging these fundamental services may consume a lot of time and effort. All of these will become an obstacle to the adoption of edge computing.

To help developers implement efficient edge applications with reasonable programming effort, we propose an edge framework called EdgeEngine. EdgeEngine contains two parts, a middleware and a programming interface. The middleware runs on top of the underlying edge infrastructure, including the end devices, the edge servers and the back-end cloud server. The programming interface is Java-based, through which developers build their edge applications. The design philosophy of EdgeEngine is that developers only focus on the business logic of their applications, while the fundamental services, including those mentioned above, are provided by the system-wide libraries and can be directly utilized without any programming effort. Meanwhile, developers can also customize the fundamental services through the programming interface in case they want to implement different policies, usually with only several lines of code.

Developers implement edge applications in *workflow*s, which can be viewed as a combination of data and the computations on the data. Workflows are hosted in Docker containers during execution, and EdgeEngine supports live migration of the workflow tasks by adopting a simple yet effective method of refining the programming practice of the developers via the programming interface. Five distributed system components are executed beneath the workflow tasks, which fulfill the fundamental services and interact with the workflow tasks to apply the developer-customized policies. The five system components are System Monitor, Data Dispatcher, Sync Manager, Transition Scheduler and Lock Manager. They are responsible for hardware tracking, data dispatching, data synchronization, task scheduling, and lock management, respectively.

We implemented a prototype of EdgeEngine and deployed it on the testbed containing five PC servers. The prototype was implemented partially based on the OpenStack project [75], for Docker management, and partially using Python and Java. A series of experiments on the prototype reveals that EdgeEngine indeed allows developers to implement edge applications with minimal programming effort and works quite efficiently under various conditions.

1.0.3 Nomad: Achieving Fast Consensus in Edge Computing

In this project, we aim to build system support to help edge applications achieve better performance. In particular, we design an efficient consensus protocol for edge computing environments, because we consider that consensus is the most fundamental problem in distributed computing while edge computing falls into the category of distributed computing. We further envision that the most common use of a consensus protocol in an edge environment is to *order* all the messages received across the entire system. Consider the following example: in an IoT payment system, customers make digital payments through their mobile devices which wirelessly connect to the nearest IoT devices. The payment messages need to be sequenced in a global order to guarantee the correctness of the financial operations. Suppose Alice has one dollar in her account. At location A, she spends the one dollar for some goods. Meanwhile, at location B, her helper spends the same dollar for some other goods. If the payment messages are not globally ordered, the IoT devices at both location A and location B may accept the payment, leading to the situation that Alice spends the one dollar twice. Therefore, the IoT payment system needs to order the payment messages so that it can accept only the first payment, while rejecting the second payment.

Being hardware-constrained, end devices typically have a limited wireless connection range, and the performance of their wireless network is relatively poor [26, 21]. In contrast, an edge computing network usually possesses very efficient interconnections, which can be utilized by the end devices for quickly ordering the messages. More specifically, the edge servers are deployed covering all the end devices and are inter-connected through highly efficient wired or wireless network links. Each end device is connected to the nearest edge server and keeps forwarding the messages it has received from the users. The edge servers communicate with each other upon receiving these messages and determine the global order upon them. This ordering process should be done as quickly as possible because all the application scenarios that we can imagine, like the IoT payment system, require minimal user-perceived latency to guarantee satisfactory user experience. This is also the reason why edge computing is a necessity here. If the user-perceived latency is not an issue, a remote cloud server would be sufficient enough for ordering all the messages.

To this end, we devise a consensus protocol on the edge computing network, called Nomad, to order the messages received across the system in a timely manner. Nomad is based on the Paxos algorithm [53], which guarantees low latency and strong consistency during the ordering process. We also introduce a series of improvements into Nomad to speed up the ordering process as much as possible. Our evaluation of a prototype reveals that Nomad works as effectively and efficiently as expected.

1.0.4 Overview

The remainder of this dissertation is structured as follows: Chapter 2 presents SMOC, a mobile-cloud/edge computing platform that performs secure computations for mobile users and allows the users to freely change the running location of the computations when necessary; Chapter 3 presents EdgeEngine, a customizable edge computing framework that allows developers to build efficient edge applications with minimal programming effort; Chapter 4 presents the design of Nomad, a consensus protocol for edge computing networks, which determines the order of the messages received across the system with as low user-perceived latency as possible; and finally, Chapter 5 concludes this dissertation and identifies several possible directions for future work.

Chapter 2

SMOC: A Secure Mobile-Cloud/Edge Computing Platform

2.1 Introduction

Smart mobile devices are gradually becoming the dominant daily computing platform for many people [70, 96]. While many applications today run directly on individual mobile devices, we envision a mobile-cloud/edge computing model emerging whereby individual devices run user interface software with the bulk of computation performed by a virtual machine (VM) running on a commodity cloud or a self-maintained edge environment [29, 120]. In this work, we aim to build a platform supporting free migration of apps between smart mobile devices and cloud-/edge-based hardware.

In the mobile-cloud/edge model, user input on the mobile device is transmitted to cloud/edge processes running on the VM. Results from cloud/edge processing are transformed into display content and then transmitted back to the mobile device. The mobile device and the remote VM share functionality to meet user needs. By moving heavy computational processes from the mobile device to the remote VM, the mobile-cloud/edge model improves user response time and reduces device energy consumption. It also has security advantages, e.g., sensitive data can be stored on the cloud/edge, safeguarded from a compromised mobile device OS or app, and also protected from exposure to a thief. A user could acquire a new smart mobile device, download the interface software, and resume arbitrary tasks from before the compromise or theft occurred.

There are already solutions for the mobile-cloud/edge model in recent literature. We believe, however, that our platform has greater potential than what is immediately obvious and can achieve more than what existing solutions can do. There are two concepts underlying our platform that differentiate it. First, we are proposing a resource-sharing platform in the sense that an app can freely change its executing location between the mobile device and the cloud/edge. In contrast, existing solutions only allow apps to run on the cloud/edge. Second, our platform provides security guarantees even when the mobile device operating system has been compromised, which is a feature that existing solutions cannot offer.

We achieve the first concept by running a VM on the cloud/edge which has an execution environment compatible with that on the smart mobile device. Our platform shares resources between the VM and the mobile device in both directions, i.e., the mobile device shares its files and I/O devices with the VM when the app is running remotely, and the VM shares its files with the mobile device when the app is running on the mobile device. The remote VM does not need to share its (virtual) I/O devices with the mobile device in the latter case, because they are not involved in the app's execution. Our system currently only supports offline migration, i.e., when an app wants to change its executing location, our platform will cease its execution if it is running, transfer all the related data to the target location, and re-launch the app if necessary. We leave online migration to future work.

Considering the first concept, our platform is clearly different from most existing solutions, such as Chrome OS [111]. Like many others, Chrome OS follows a client-server computing model in which the cloud behaves as a server that hosts apps, while the smart mobile device behaves as a client that communicates with the cloud. This model fails to meet a wide range of user needs, because it only allows apps to run on the cloud. Our platform, on the other hand, is a resource-sharing platform, in which apps can run on both locations and can freely change their executing location.

The second concept is achieved by leveraging the hardware virtualization functionality on the smart mobile device. To be more specific, we suppose that a hypervisor runs on the smart mobile device, which hosts a guest OS. The guest OS could be malicious, and may launch attacks on the apps that it hosts. The apps may also be malicious, compromised, or attacked by the guest OS. In any case, they may leak sensitive information stored on the smart mobile device or input by the user. However, we trust the hypervisor, because the hypervisor is always much smaller than the guest OS, and can be fully verified through formal verification or manual audit. Moreover, the hypervisor is unlikely to install any third-party applications or libraries, and thus gets rid of many potential risks. We also trust the remote side, which is either a commodity cloud or a self-maintained edge. In the former case, we assume that the cloud is established by a famous company with high concern for their reputation, such as Amazon, Google, Apple and Microsoft. These companies usually have the technical strength to protect their clouds, and are unlikely to intentionally compromise user privacy. In the latter case, we assume that the edge is maintained by the user herself, who cares about security and runs anti-virus software on the edge server(s), or even cuts down the edge's network connection to the outside and dedicates the edge server(s) to the mobile-edge platform.

Under these assumptions, our platform offers security guarantees concerning an untrusted guest OS running untrusted apps. Our novel approach is to make the hypervisor responsible for sharing the smart mobile device's input interfaces with the cloud/edge, and for blocking hardware input events from traveling to the guest OS. By doing this, we can guarantee that the guest OS can learn nothing about the user's input, while at the same time remain responsible for sharing the smart mobile device's output interfaces with the cloud/edge. To the best of our knowledge, we are the first to leverage hardware virtualization to support security reinforcement for smart mobile devices.

The following example sheds light on this idea. Consider a smart mobile device, running a compromised guest OS, which executes a malicious background service that stealthily records the user's input through a software keyboard on the screen and sends it to a remote, malicious user. In this case, if the user runs a banking app directly on the smart mobile device, or through any existing mobile-cloud/edge solution, her account and password are likely to be leaked to the malicious user. If the user runs the banking app on our platform, however, there is no such security concern. The hypervisor will capture the touch events from the software keyboard and forward them directly to the remote VM, bypassing the guest OS entirely. Meanwhile, the banking app functions properly on the remote VM, because it receives the user's input from the hypervisor.

To summarize, our contributions in this work are three-fold.

- We propose a secure mobile-cloud/edge computing platform which is designed as a resource-sharing platform in the sense that apps running on it can freely change their executing location. This is a more flexible design than those of existing solutions, and it meets a wider range of user needs.
- We are the first to leverage hardware virtualization on the smart mobile device to provide security guarantees in the case that the smart mobile device's operating system is untrusted.
- We implement a prototype system following our platform design, and conduct several experiments on this system. The results demonstrate that our platform is efficient and practical.

2.2 Related Work

Mobile offloading. An increasing amount of attention has been invested in mobile offloading recently. CloneCloud [24, 23] partitions a mobile application automatically by using static analysis and dynamic profiling, such that part of the application can be offloaded to the cloud to achieve performance and an energy efficiency improvement. It also provides a runtime system to facilitate the offloading. MAUI [27] also approaches the topic of mobile offloading from the perspective of automatic mobile program partition. The main objective of the partition is to dynamically decide which part of the program should be offloaded to achieve maximum energy savings. COMET [39] implements a runtime system on top of the Dalvik Virtual Machine, the virtual machine used by Android, to allow for simultaneous executions of the same multi-threaded application in several machines. The enabling foundation is a distributed shared memory model. In contrast with these existing works, our platform is a resource sharing platform through which the user can run an app on either the cloud or the smart mobile device.

Thin client architecture in mobile computing. When a user decides to run a mobile application in the cloud with our system, we essentially turn the local mobile device into a thin client. MobiDesk [10] introduces a mobile virtual desktop hosting infrastructure which transparently decouples the display, the operating system and the network of a user's computing session from end-user mobile devices, and moves them to hosting providers. SmartVNC [99] is a system to port VNC, which is a remote computing solution, to smartphones while achieving the same level of user experience as on PCs. While work like MobiDesk and SmartVNC try to reduce workloads in mobile devices by turning the devices into thin clients, other works have also studied the energy consumption implication of applying the thin client architecture in mobile computing [67, 58]. Distinct from these works, our platform adopts a spilt client design, combined with hardware virtualization on the smart mobile device, to provide security guarantees even if the smart mobile device's guest OS is untrusted.

Exploiting hardware virtualization technology. Hardware virtualization technologies on the x86 architecture [100, 3] have long been used to develop solutions to protect system security. For example, SecVisor [90] utilizes AMD-V [3] (formally known as AMD Secure Virtual Machine (SVM)) to build a hypervisor with a small code base to ensure the code integrity of guest OS kernels. Specifically, the hypervisor by SecVisor is able to prevent code injection attacks by allowing only user-approved code to run in kernel mode. TrustVisor [64] also uses AMD-V to build a small hypervisor to ensure the guest OS's data integrity and secrecy, in addition to code integrity. Lares [79] exploits Intel VT-x [100] to achieve active security monitoring inside a virtual machine environment. In our platform, we choose to allow the mobile device OS and apps to run on VMs in the cloud. Part of the reason for this choice is to utilize these existing solutions to ensure we have a secure execution environment in the cloud. On the local device side, we exploit the newly developed ARM hardware virtualization technology [68] to protect the user's input and data collected by various on-board sensors, which can reveal sensitive information about the user. Although there are some efforts to use this technology to build general purpose hypervisors [30, 103], to the best of our knowledge, we are the first to utilize it to build a system specifically for security/privacy.

2.3 Platform Design

Our platform spans across the user's local mobile device and a remote cloud server. We believe that such a mobile-cloud computing platform is more capable than what people usually think and can achieve more than what existing solutions can do. In this section, we describe our platform design in detail.

2.3.1 Design Goals

The main design goal of our platform is to share the necessary resources between the smart mobile device and the cloud, such that an app can run either on the smart mobile device or in the cloud, arbitrarily. This design goal contains two aspects. First, the platform must be capable of executing an app in the cloud without any modifications to the app itself. This is possible only if the platform shares the smart mobile device's I/O interfaces with the cloud, such that an app running in the cloud can receive the user's input (e.g., from touch screen, sensors, keyboard, etc.) and render the output (e.g., display, sound, etc.) on the mobile device. Second, an app installed in the cloud should be able to be downloaded to the smart mobile device and work properly, as it does in the cloud. This can be achieved if the app can access the same resources regardless of its location. For example, files accessed by the app should be synchronized across both locations.

Running apps in the cloud is necessary for several reasons. First, the user may not trust the app (e.g., because the app has access to sensitive data) but may still want to run it. In this case, she may utilize our platform and run the app on a VM in the cloud. Even if the app compromises this VM, it cannot affect other apps and data on the user's local mobile device, and the user may simply delete the VM afterwards. In this way, even a malicious app is completely quarantined. Moreover, commercial clouds often provide powerful anti-virus services, and a malicious app is more likely to be caught if these services are implemented in the cloud used in our platform. Second, smart mobile devices are always resource constrained [41, 117, 116]. To solve this problem, our platform allows the user to offload apps from the smart mobile device to the cloud, which typically has abundant resources. Third, some organization or developer may want to publish an app without disclosing any proprietary secrets about it (e.g., the binary of the app might be reverse engineered to compromise some critical algorithms). In this case, the organization/developer may publish the app using our platform and only allow the app to run in the cloud. In this way, no malicious user can recover a complete binary file for further analysis.

After being deployed, our platform will have some apps installed in the cloud by default. However, the user may want to download an app from the cloud to the smart mobile device for some reason, for example, the latency between the smart mobile device and the cloud becomes unacceptable. This can be achieved transparently in our platform, because all resources, including files, are shared between the smart mobile device and the cloud.

The cloud exploited in our platform is considered a commercial cloud. However, it could also be a private cloud established by, and serving, only a single user. Consider the scenario where a user does not trust an app, but still wants to run it. She may build a private cloud which does not connect to the Internet, and deploy our platform on this cloud in conjunction with her smart mobile device through a local area network. She can then run the app in the private cloud to prevent her mobile device from being attacked by the app.

Another design goal of our platform is to provide a secure environment for the user to run apps. As we mentioned previously, the VM in the cloud is isolated from the smart mobile device, and thus a malicious app running on the VM cannot affect any other apps running on the smart mobile device. Moreover, commercial clouds often provide powerful anti-virus services, which can be utilized on our platform to defend against malicious apps. These are two cases in which our platform can provide strong security guarantees. We have yet to consider another important scenario; suppose a user wants to run a banking app on the smart mobile device, but the smart mobile device's operating system has been compromised and a malicious background service stealthily records the user's input and sends it to a malicious user. In this scenario, running the banking app on the smart mobile device is dangerous because the malicious user may learn the user's banking account and password via her input. Our platform provides a secure environment that defends against this attack by leveraging hardware virtualization on the smart mobile device.

2.3.2 Design Details

Figure 2.1 demonstrates our design when an app is running in the cloud on our platform. The app is running in the cloud, hosted by a VM which has an execution environment compatible with that of the smart mobile device. This allows the app to run without requiring any changes. Hardware virtualization is enabled on the smart mobile device. A hypervisor running on the smart mobile device hosts one or more guest OSes in which the app can be executed.

As described in our main design goal, the smart mobile device needs to share I/O

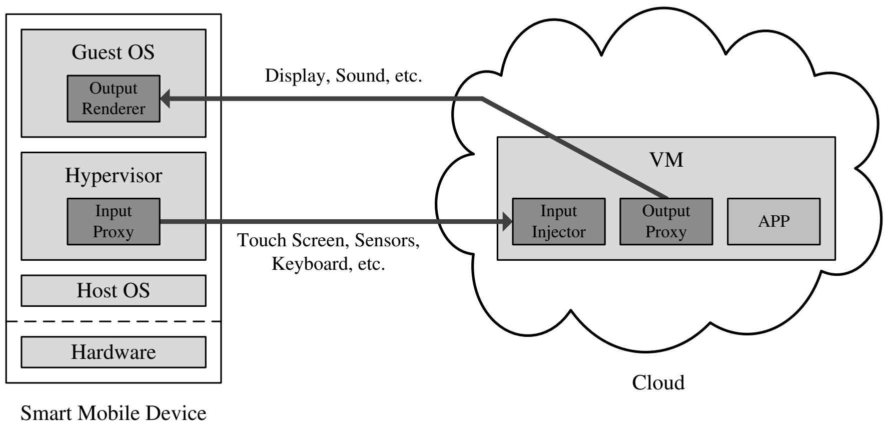


Figure 2.1: An app is running in the cloud in our platform.

interfaces with the VM such that the app can work properly in the cloud. To achieve this, two client programs are provided on the smart mobile device. One of them, depicted as “Input Proxy” in Figure 2.1, is responsible for capturing the user’s input through the smart mobile device’s input interfaces (e.g., touch screen, sensors, keyboard, etc.) and sending it to the VM. The VM will then emulate the user’s input such that the app can receive it and work properly. On the other hand, the app generates output (e.g., display, sound, etc.) when it is running in the cloud and sends the output to the VM’s output interfaces. The VM will then transfer the app’s output to the smart mobile device. The other program, depicted as “Output Renderer” in Figure 2.1, is responsible for receiving this output and rendering it on the smart mobile device so the user can perceive it. By communicating in this way, the smart mobile device shares its I/O interfaces with the cloud VM such that the app can be executed in the cloud instead of on the smart mobile device.

Figure 2.2 demonstrates our design in the case that an app installed in the cloud is downloaded to, and executed on, the smart mobile device. Our platform allows the user to either launch the app in the cloud or download it to the smart mobile device and execute it. When the user downloads the app to the smart mobile device, a distributed file system across the VM that hosts the app in the cloud and the guest OS on the smart mobile device will be automatically enabled. This distributed file system is needed in order for the VM to

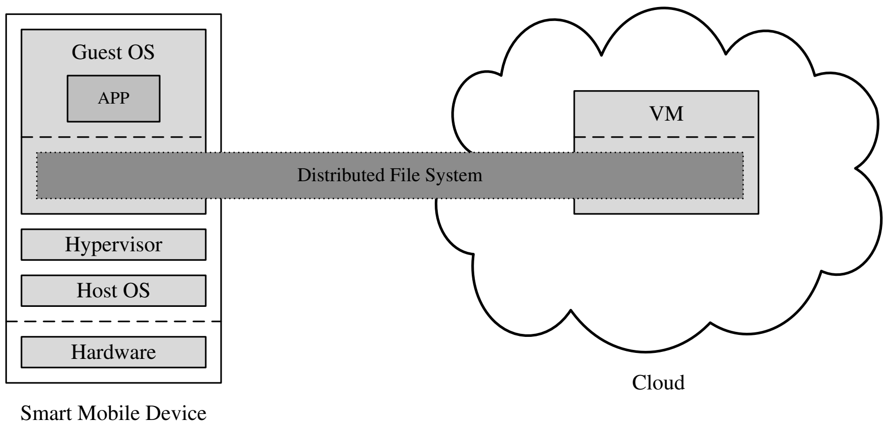


Figure 2.2: An app installed in the cloud is downloaded to, and executed on, the smart mobile device in our platform.

share resources with the smart mobile device, allowing the app to work properly without any modifications. Recall that the smart mobile device needs to share its I/O interfaces with the VM when the app is running in the cloud. As the app is now running on the smart mobile device, it can directly access the smart mobile device's I/O interfaces, so no I/O interface sharing is needed. However, the app, previously installed in the cloud, may need to access some resources (such as files) on the VM to work properly. For this reason, we design the distributed file system in our platform.

There are two ways to download an app from the cloud to the smart mobile device. The first way is offline downloading, in which the user downloads the app's binary file when the app is not running, and executes it on the smart mobile device thereafter. The second way is online downloading, in which the user migrates the app's process while the app is running, and continues its execution on the smart mobile device. For simplicity, our platform currently only supports offline downloading. However, conceptually, online downloading could also be achieved. We leave this to future work. After being downloaded to the smart mobile device, the app may need to access files on the VM that previously hosted it. Our distributed file system, spanning across the smart mobile device and the VM, allows the app to access such files after it has been downloaded. Therefore, the app can work properly on the smart mobile device without any modifications.

Combining these two design elements, illustrated by Figure 2.1 and Figure 2.2, our platform achieves its main goal. The second design goal can also be achieved if the hypervisor in Figure 2.1 is trusted. This is a reasonable assumption, because the hypervisor is always much smaller than the guest OS, and can be fully verified through formal verification or manual audit. Moreover, the hypervisor is unlikely to install any third-party applications or libraries, and thus gets rid of many potential risks.

As we mentioned previously, the guest OS running on the smart mobile device is untrusted in our platform. It might be malicious and runs a stealthy key logger in the background. In this case, the user's private information, such as the banking account and password, could be compromised. To solve this problem, our platform leverages hardware virtualization on the smart mobile device and adopts a split design for the client program to share the smart mobile device's I/O interfaces. Figure 2.1 demonstrates how our platform provides strong security guarantees in the case that the guest OS is untrusted. The guest OS runs on the trusted hypervisor, rather than directly on the smart mobile device's hardware. The app runs on a VM in the trusted cloud. The input proxy, residing in the hypervisor, traps the user's input such that it cannot be received by the guest OS, and transfers it to the VM. This is feasible because the hypervisior is the first layer on the smart mobile device at which hardware events (such as touch points) arrive, so the input proxy can process these events before passing them to the guest OS, or hiding them from the guest OS. Therefore, the app can work properly in the cloud, but the guest OS cannot learn anything about the user's input. On the other hand, the guest OS is the location where the user "logically" launches the app, so it should also be the location where the app's output should be rendered. For this reason, the output renderer is placed in the guest OS, responsible for rendering the app's output.

Our platform provides security guarantees even if the guest OS is untrusted, because the VM in the cloud takes the place of the untrusted guest OS. The app runs on the VM instead of the guest OS, and the user's input is sent to the VM but hidden from the guest OS. However, this implies that our platform only works if the cloud is trusted. We do trust the cloud in our platform, as we mentioned previously, by assuming that the cloud is provided by a famous company with high concern for their reputation, such as Amazon, Google, Apple and Microsoft. These companies are unlikely to intentionally compromise their users' privacy. Moreover, their clouds often provide powerful anti-virus services, and thus a compromised VM or app running in the cloud will likely be caught by these anti-virus services.

It is also arguable that the user may delete the guest OS if she suspects the guest OS has been compromised, and install a new one that is safe to execute apps. This solution may also solve the problem, but has some drawbacks. The main drawback results from the fact that a smart mobile device is usually resource constrained and thus can support only a limited number of guest OSes. In this case, a guest OS on the smart mobile device may have many apps installed and most of them may have no security issues, even if they are executed in a compromised guest OS. Therefore, the user may be unwilling to delete this guest OS. Meanwhile, the limited resources on the smart mobile device prohibit the user from installing a new guest OS.

The second design goal can only be archived by the design illustrated in Figure 2.1, which implies that the user needs to execute the app in the cloud to enjoy strong security guarantees. When the user downloads the app to the smart mobile device for some reason, e.g., the latency between the smart mobile device and the cloud becomes unacceptable, she implies that she wants to trade strong security guarantees for higher usability. Therefore, it is natural that our platform does not make the same security guarantees in this case.

2.4 System Implementation

To prove that our platform is practical and can work well, we implement a prototype system, following the design outlined in the previous section. In this section, we elaborate on the implementation details of this prototype system.

2.4.1 Setup

The smart mobile device used in the prototype is a Samsung Chromebook. We choose this mobile device because it is one of a few that support hardware virtualization, which is required by our platform. The host OS installed on this Chromebook is Ubuntu Linux and the hypervisor running on this host OS is KVM plus QEMU. Both KVM and QEMU are customized for the Chromebook, as provided by [104], the authors of which also provide a bootloader to enable the hardware virtualization features of the Chromebook. QEMU emulates a Cortex-A15 VExpress hardware abstraction, and Android is installed on this hardware abstraction as the guest OS.

The cloud in the prototype system is established on a Lenovo laptop. The host OS on the cloud is Windows, and VMware Workstation is installed on the host OS as the hypervisor. A VM is created on the VMware Workstation with Android-x86 [6] installed as the guest OS.

2.4.2 Sharing I/O Interfaces

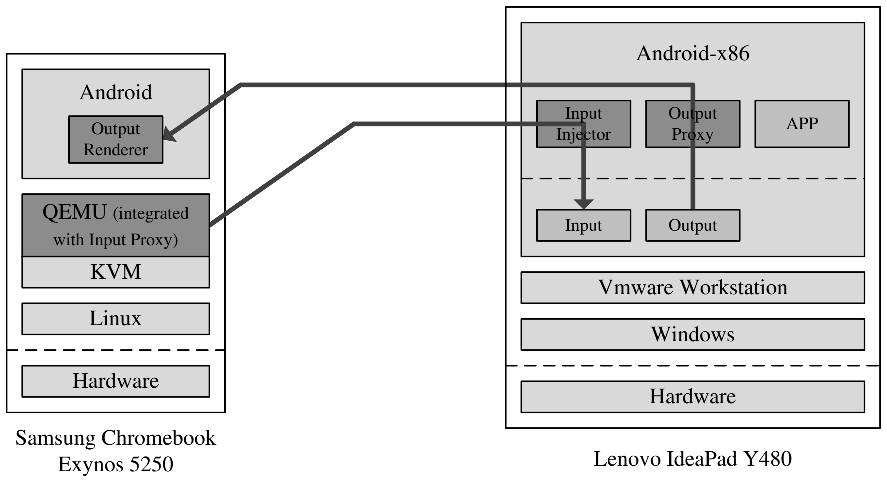


Figure 2.3: An app is executed in the cloud in the prototype system. Darker blocks indicate the components where our implementation has been involved.

Figure 2.3 demonstrates how the components cooperate in the prototype system when an app is executed in the cloud. Darker blocks indicate the components where our implementation has been involved. As depicted in Figure 2.3, four components are implemented to share the I/O interfaces across the smart mobile device and the cloud. They are input proxy, input injector, output proxy and output renderer.

The input proxy works with the input injector to share the input interfaces across the smart mobile device and the cloud. As described in Section 2.3.2, the input proxy resides in the hypervisor on the smart mobile device. We integrate the input proxy into QEMU in our implementation, because QEMU is part of the hypervisor. More specifically, we implement the input proxy's functionality in QEMU, such that QEMU can capture the user's input from the keyboard (i.e., the hardware keyboard) as well as an attached accelerometer (SEN-10537 Serial Accelerometer Dongle), and transfer it to the cloud. There is not much difference between the keyboard and the accelerometer from QEMU's point of view. Therefore, we only focus on the keyboard part in the following discussion.

As the input proxy is integrated in QEMU, we need to define a way by which QEMU can enter and exit the input proxy mode to emulate the input proxy's launch and termination. In our implementation, QEMU will enter the input proxy mode when it detects that the user has repeatedly pressed the "Caps Lock" key six times. After entering the input proxy mode, QEMU will continue monitoring the keyboard and exit this mode if it receives an ESC keystroke. When it is in the input proxy mode, QEMU maintains a connection to the cloud. Upon receiving a key event from the host OS, QEMU will encrypt the key code and send it to the cloud. It will not generate the virtual keyboard interrupt for the guest OS. Therefore, the guest OS will not know this key event and thus cannot learn the user's keyboard input, as described in Section 2.3.

We have considered several places to implement the input proxy functionality. The first place we have considered is the host OS's keyboard driver. However, we have quickly found that this is not a good choice, because it affects all the programs residing in the host OS alongside the hypervisor (i.e., any program running on the host OS will not receive any input events from the keyboard). The second place is the guest OS's keyboard driver.

We have also decided against this choice, because it violates our platform design by making the guest OS capable of learning the user's input when the app is running in the cloud. According to our design in Section 2.3, the most natural place to implement this functionality is in the hypervisor. As QEMU is responsible for providing the hardware abstraction to the guest OS, we have decided to implement this functionality in QEMU.

The input injector in the cloud, implemented as an Android app on the Android-x86 VM, is responsible for maintaining the connection to QEMU. It also listens on this connection for encrypted key codes sent by QEMU. When an encrypted key code is received, the input injector will decrypt it, and inject the key code into the VM through the Linux *sendevent* utility. Then the app running on the VM will be notified of the key event and receive the key code.

The output proxy works with the output renderer to share the output interfaces across the smart mobile device and the cloud. We only consider sharing the screen frames in our prototype system, as this is enough for the user to track the app's output on the mobile device in most cases. Therefore, the output proxy needs to take a screenshot of the Android-x86 VM periodically and transfer it to the output renderer, which will then render it on the smart mobile device's screen.

The output proxy is implemented as another Android app alongside the input injector on the Android-x86 VM. It is responsible for maintaining a connection to the output render, and for sending the VM's screenshots periodically, as fast as possible, through this connection. It captures the VM's screenshot by invoking the *screencap* program provided by Android-x86. A captured screenshot is then stored as a png image file on the local disk. Finally, the output proxy sends the file data to the output renderer, which will further display the screenshot on the smart mobile device's screen.

The output renderer is implemented as an Android app running in the Android guest OS on the smart mobile device. It connects to the output proxy when the app is launched in the cloud. After the connection has been established, the output renderer listens to this connection for the VM's screenshots sent by the output proxy. When a screenshot is received, the output renderer stores it as a local png image file and sets this file as the source of its `SurfaceView` component. The `SurfaceView` component then displays the image file on the screen. This process is done periodically, as fast as possible, such that the user will perceive the stream of the app's screen output as if the app were running locally.

2.4.3 Distributed File System

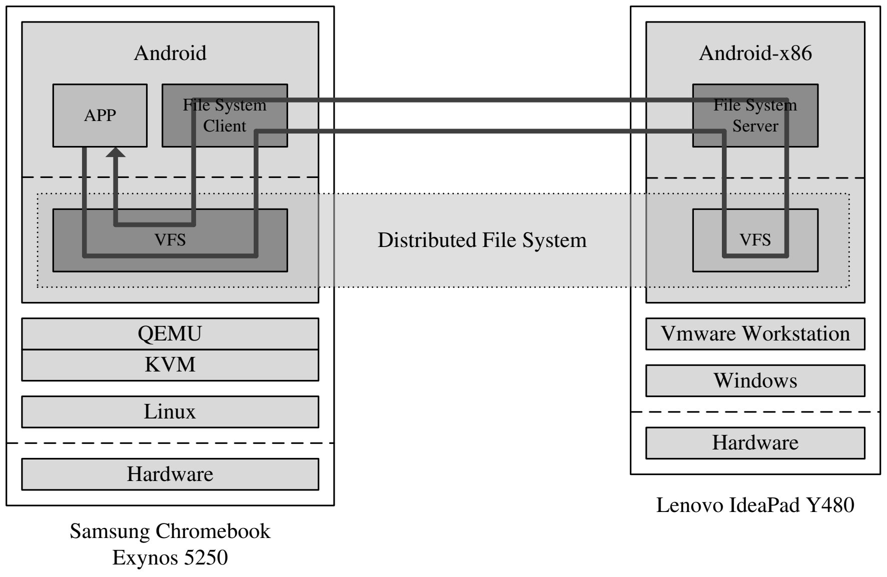


Figure 2.4: An app, installed in the cloud, is downloaded and executed on the smart mobile device in the prototype system. Darker blocks indicate the components where our implementation has been involved.

Figure 2.4 demonstrates how the components cooperate in the prototype system when an app, installed in the cloud, is downloaded and executed on the smart mobile device. As discussed in Section 2.3.2, we need to share files between the cloud VM and the smart mobile device to allow the app to work properly on the smart mobile device. To achieve this, we implement a distributed file system across the smart mobile device and the cloud in our prototype system.

The implementation of the distributed file system contains three parts, as illustrated in

Figure 2.4. First, the VFS in the Android guest OS is modified such that it communicates with a user-level program depicted as "file system client" in Figure 2.4. This is achieved by leveraging the netlink socket mechanism provided by Linux, as Linux is the underlying kernel of Android. Through a netlink socket, a user-level program and the Linux kernel can communicate with each other in a straightforward manner. Second, the file system client is a user-level program running in the Android guest OS on the smart mobile device. This program is implemented as a Linux binary responsible for communicating with both the local Linux kernel and the cloud. Third, the "file system server" in Figure 2.4 is a user-level program running on the Android-x86 VM in the cloud. This program is also implemented as a Linux binary, responsible for communicating with the file system client on the smart mobile device and executing file operations on the local file system.

As illustrated in Figure 2.4, the smart mobile device and the cloud communicate with each other through a connection between the file system client and the file system server in the distributed file system. This can also be implemented in several ways, e.g., the VFS on the smart mobile device may directly communicate with the file system server, or even with the VFS on the VM, to avoid switching between the kernel and the user space. We choose to use user-level programs to manage this connection, because they are more robust and easier to configure, even though they introduce some overhead when interacting with the local VFS in the kernel.

When an app is executed in the cloud, it may only access local files. To access a local file, it will first invoke the system call *sys_open()* to communicate with the VFS, which will open the file and return the file descriptor to the app. Through this file descriptor, the app can read data from the file, write data to the file, or close the file by invoking the *sys_read()*, *sys_write()* and *sys_close()* system calls, respectively. When the app is downloaded from the cloud and executed on the smart mobile device, however, it may access remote files in the cloud. The distributed file system is implemented to support this kind of remote file access in the prototype system.

To implement this distributed file system, we begin with a simple solution. We redirect every *sys_open()*, *sys_read()*, *sys_write()* and *sys_close()* system call that is invoked on the smart mobile device's VFS to the VM's VFS running in the cloud. To be more specific, when an app running on the smart mobile device tries to open a file that is considered in the cloud (e.g., in a directory under */sdcard*), the smart mobile device's VFS will switch to user space by notifying the file system client of this event. Then the file system client will redirect the *sys_open()* system call to the file system server in the cloud with exactly the same parameters. After receiving this redirected system call, the file system server invokes it on the VM's VFS and gets the corresponding file descriptor. It then returns this file descriptor to the file system client, which will switch back to the local VFS with the file descriptor it has received. The VFS logs this file descriptor as a remote one and returns it to the app. Then the app can use this file descriptor to read, write and close the remote file. When the app reads the file through this file descriptor, the VFS will notice that this file descriptor actually refers to a remote file. It will then redirect the *sys_read()* system call to the VM's VFS similarly to when it redirects the *sys_open()* system call.

The solution described above is straightforward. However, it involves too many network communications between the smart mobile device and the cloud, and too frequent switching between the user space and the kernel on the smart mobile device. As a result, it has prohibitively low performance. To solve this problem, we implemented the distributed file system differently. Figure 2.4 illustrates part of this new solution, i.e., how it works when the app tries to open a remote file in the cloud. When receiving a *sys_open()* system call from the app and finding that it wants to open a file in the cloud, the smart mobile device's VFS will switch to user space by notifying the file system client of this event. The file system client then communicates with the file system server, which will send back the data of the corresponding file. The file system client then stores the file on the local disk as a cached file, with a temporary file path that is not likely to conflict with that of any other file. Then the smart mobile device's VFS opens this file instead of the remote file and returns its file descriptor to the app. The file descriptor is marked as "remote" and the remote file path is recorded in the file descriptor by using two fields that we have added in the file descriptor data structure. Then the app can use this file descriptor to read and write the cached file. When closing the file, the VFS will notice that it is a cached file. It will switch to user space by notifying the file system client of this event. The file system client then sends the cached file back to the file system server, which will overwrite its local file with the cached file in the cloud.

It is arguable that this solution may cause inconsistency when an app on the smart mobile device and an app in the cloud access the same file simultaneously. Nevertheless, it is rare that two apps access the same file simultaneously. Most mobile OSes, such as Android, forbid or at least recommend against an app accessing files belonging to another app, for security purposes. Furthermore, although it is not mandatory, a prudent way to use our platform is to create a separate VM in the cloud for every app. Two apps will never access the same file simultaneously in this case.

2.5 Discussion

Hypervisor in local mobile device. To allow for consistency with our system implementation, we present our system design in a way that used a "Type 2" hypervisor (i.e., a hosted hypervisor that runs in a host OS) in the local mobile device to achieve high security and privacy guarantees. It is worth noting that it is not necessary to use Type 2 hypervisor in our design. Actually, a Type 1 hypervisor (i.e., bare-metal hypervisor that runs directly on top of the hardware) fits better into our goal of providing high security guarantees. This is because the main reason we use a hypervisor in local mobile device is to isolate the sensitive inputs (i.e., user taps on the touch screen, data collected by various on-board sensors) from local mobile device OS and apps. These components are greater risks if they are compromised. A Type 1 hypervisor, with a small trusted computing base (TCB), can also fulfill our needs, but developing a new hypervisor from the ground up, based on the newly introduced ARM hardware virtualization, would require a lot of effort in engineering optimal parameter settings. Therefore, we opt to use KVM, which is a full-fledge hypervisor that has been recently ported to the ARM architecture, for a fast proof of concept demonstration of our system design. For future work, we plan to develop our own bare-metal hypervisor to further improve our system.

The use of Chromebook in the prototype system. We use a Samsung Chromebook as the mobile device in our prototype system implementation. The main reason for this choice is that it has a hardware configuration that supports ARM hardware virtualization, and it requires relatively less effort to enable and run the ARM based KVM, which is the base hypervisor for our prototype system. Although the Samsung Chromebook looks more like a regular laptop, it actually shares the hardware similar to most recent smartphones and tablets. For example, the Chromebook used in our system has a Samsung Exynos 5 Dual SoC, which is the same SoC found in the Google Nexus 10 tablet. The ARM Cortex-A15 MPcore processor contained in the Exynos 5 Dual SoC is also used in many other smartphones, such as the Samsung Galaxy S4/S5 smartphones, the Galaxy Note 3 smartphone and the Galaxy Tab Pro tablets. The 2 GB RAM capacity in our Chromebook is also the standard configuration commonly found in the latest smartphones. Therefore, our choice of using Chromebook as the mobile device can fit well into the smart mobile device world in terms of computational capability.

2.6 Evaluation

In this section, we describe the real-world experiments we have conducted to evaluate the performance of our prototype system.

2.6.1 Experiment Setup

Our prototype system consists of two parts, the smart mobile device and the cloud VM. The smart mobile device is a Samsung Chromebook featuring the Exynos 5 Dual SoC, 2 GB RAM and 16 GB SSD hard drive. The Exynos 5 Dual SoC is equipped with a 1.7 GHz dual-core ARM Cortex-A15 processor. The ARM Cortex-A15 processor has hardware virtualization support, which allows us to implement the proposed hypervisor in our prototype system. The Chromebook runs Linux (kernel version 3.13.0) as the host OS. On top of the hypervisor, we run Android Jelly Bean (Android version 4.1.1, Linux kernel version 3.9.0) as the guest OS. The cloud VM runs Android-x86 [6] as the guest OS (Android version 4.3, Linux kernel version 3.10.2). It is hosted by the VMware workstation 10.0.1 virtual machine monitor (VMM) running on a host PC with an Intel Core i7 CPU (2.3 GHz) and 8 GB RAM. The smart mobile device and the cloud VM are connected through an 802.11n wireless router.

2.6.2 Running Apps in the Cloud: The Response Time

The most significant feature of our system is that we allow the entire mobile device OS and the apps to run in the cloud. Therefore, we first evaluate the app response time when the apps are running in the cloud. In our system, we send the output of an app running in the cloud back to the device by first taking screenshots of the app, and then transmitting them back to the device. Therefore, there are two major factors that can affect app response time, network bandwidth and the size of each screenshot. We design an experiment to evaluate the impacts of these two factors. In our experiment, we use an image viewer app to open different pictures. Remember that with our system, the app launch and the picture opening operations are triggered by the user on the mobile device side, and the actual operations are conducted on the cloud side. Once a picture is opened, our system takes a screenshot of it, and sends the screenshot back to the mobile device to display to the user. We measure the response time as the time difference between the point when the user triggers the picture opening action on the mobile device, and the point when the screenshot is sent back from the cloud and displayed. We choose different pictures, such that we have screenshots with different sizes (14 KB, 82 KB, 178 KB, 236 KB, 392 KB and 512 KB are used in our experiment). We also configure the wireless router to achieve different network bandwidth rates (0.25 Mb/s, 0.5 Mb/s, 1 Mb/s, 2 Mb/s, 10 Mb/s and 100

Mb/s are used in our experiment).

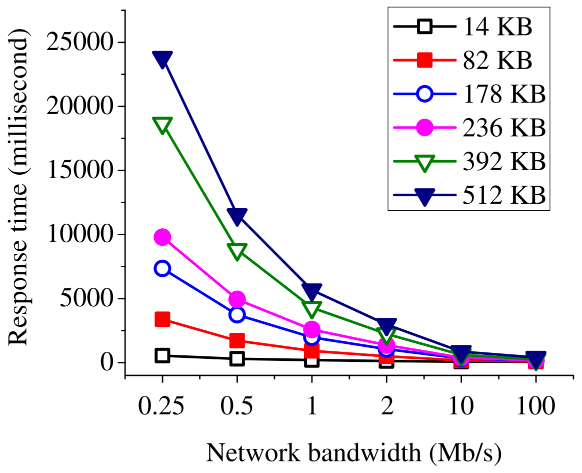


Figure 2.5: App response time if running in the cloud: UI response time (Y-axis) of screenshots with different size under different network conditions (X-axis).

Figure 2.5 depicts the experiment results. We can see that network bandwidth has a significant impact on app response time when the bandwidth is small. But this impact gradually diminishes as the available bandwidth increases. For example, when the network bandwidth is fixed at 0.25 Mb/s, it takes only 0.5 second to open a picture 14 KB in size. But it needs almost 24 seconds to get back a 512 KB screenshot. The response time ratio for these two screenshot sizes (i.e., $\frac{512\text{KB}}{14\text{KB}}$) is $\frac{24\text{second}}{0.5\text{second}} = 48$. But when the network bandwidth is set at 100 Mb/s, the response time ratio for the same screenshot sizes (i.e., $\frac{512\text{KB}}{14\text{KB}}$) is only $\frac{0.4\text{second}}{0.07\text{second}} = 5.7$. Fortunately, modern mobile data networks can support a very high transmission rate. The 4G LTE network has a peak download speed approaching 100 Mb/s [110], and on a typical day 4G download speeds can range from 2.8 Mb/s to 9.1 Mb/s, with an average value of 6.2 Mb/s [126]. With the average 4G download speed (6.2 Mb/s), the app response time of our prototype system for a screenshot of 512 KB is only about one second.

Table 2.1: Hypervisor overhead.

	Native OS	**With unmodified hypervisor**	**With our hypervisor**
Delay	4 ms	63 ms	65 ms

2.6.3 Performance of the Hypervisor on the Local Device

Our system exploits the ARM hardware virtualization technology to achieve user/sensor input isolation from the mobile device OS. Specifically, when an app is running in the cloud, our hypervisor intercepts all the inputs from the user (e.g., touch screen, keyboard, etc.) and sensors, performs an encryption on them, and sends them to the cloud. We design an experiment to evaluate the overhead introduced by our hypervisor. In this experiment, we use keyboard input to evaluate the hypervisor overhead. We test three cases. In the first case, we use the keyboard to provide inputs for a user program running on the native OS of the mobile device. In the second case, we test the same scenario, except that the user program is running in the guest OS of the mobile device. In this case, the unmodified hypervisor will introduce some overhead to the keyboard input operation. The third test case shares the same setup as the second one, except that the hypervisor is the one used in our prototype system, which performs encryption on the intercepted keyboard inputs. Because we want to test the overhead caused by our hypervisor, we direct the encrypted input up to the user program in the guest OS, instead of redirecting it to the cloud. In all the three cases, we measure the time delay between the keyboard input interrupt and the time when the user program receives the input. We perform each test ten times, and report the average value here. Table 2.1 shows the results of the experiment. The results suggest that by running the user program in the guest OS, the time delay increases by one order of magnitude, compared to that when the user program is running in the guest OS. This is normal because the guest OS involves many switches between many entities including the guest OS, the hypervisor, the host OS and the QEMU hardware emulator (required by KVM). It is worth noting that when comparing to the case with the unmodified hypervisor, our hypervisor only incurs a very small amount of additional delay (about 3%).

Table 2.2: Performance of the distributed file system.

	64 KB	256 KB	512 KB	1 MB	5 MB	10 MB
Open	36 ms	85 ms	115 ms	225 ms	827 ms	1324 ms
Read	1 ms	1 ms	5 ms	11 ms	20 ms	38 ms
Write	1 ms	3 ms	6 ms	13 ms	68 ms	128 ms
Close	39 ms	68 ms	116 ms	219 ms	680 ms	1337 ms

Table 2.3: Performance of the native file system.

	64 KB	256 KB	512 KB	1 MB	5 MB	10 MB
Open	1 ms	1 ms	2 ms	5 ms	14 ms	24 ms
Read	1 ms	1 ms	6 ms	11 ms	21 ms	40 ms
Write	1 ms	3 ms	6 ms	13 ms	67 ms	126 ms
Close	1 ms	1 ms	2 ms	3 ms	14 ms	27 ms

2.6.4 Performance of the Distributed File System

The purpose of our distributed file system is to allow users to run apps locally on their mobile device. In this experiment, we evaluate the performance of file open, read, write and close operations of our distributed file system. Each test is performed ten times. Table 2.2 shows the performance results of our distributed file system. As a comparison, Table 2.3 shows the results of the native file system. From these results we can see that our file system incurs a time overhead for the open and close operations. This is because when an app running in the mobile device tries to open a certain file that is not available in the local device, our file system transparently caches the file of interest from the cloud to the mobile device to allow the app to proceed. When the app finishes accessing and closes the file, our file system automatically writes the file back to the cloud. Therefore opening files is costly. But writing and reading them is much less so. Since we are using whole file caching in our current implementation, file size and network bandwidth have major impacts on the open/close delay. For file read/write, since our file system allows local access to the cached copy, it has the same performance as the read/write operations in a native file system.

2.7 Conclusions

The mobile-cloud/edge computing model will be the dominating trend in the future. However, existing solutions do not fully exploit the potential of this model. To this end, we aim at designing a solution that goes beyond the current state of the art. In this chapter, we propose a novel mobile-cloud/edge platform with two fundamental contributions. First, our platform allows users to freely choose to run their applications either on the cloud/edge or on their local devices. We feel that this is a very useful and practical feature for users, and believe that we are the first to consider this situation in a mobile-cloud/edge platform of this kind. Second, our platform provides security guarantees against untrusted applications and an untrusted local device's operating system, by leveraging hardware virtualization technology. To the best of our knowledge, we are the first to utilize hardware virtualization to strengthen the security on mobile devices. Based on these design concepts, we build a prototype system on a Chromebook acting as the user's local mobile device, and a commodity x86 laptop PC acting as a cloud/edge server. Our evaluation on the prototype system proves that our platform is useful and pragmatic.

Chapter 3

EdgeEngine: An Efficient and Customizable Framework for Edge Computing

3.1 Introduction

Edge computing has emerged over the last several years as a new computing paradigm that extends cloud computing [87, 125, 42]. By executing tasks at the edge of network, edge computing establishes an environment that enjoys better network conditions, including shorter network latency, higher network bandwidth and more stable network connections. All of these are critical for QoS-sensitive applications, such as mobile-cloud applications, IoT applications, big data applications with real-time constraints, and so on.

Despite the benefits mentioned above, it is still quite challenging to efficiently employ edge computing in the real world. The main difficulty is that developers have to put a lot of effort into implementing applications for edge environments. They need to carefully coordinate client devices, edge nodes and the cloud in their implementation, guaranteeing that their applications could enjoy the benefits provided by edge computing. This is usually a hard work and requires that the developers have enough experience to make their implementation work efficiently on the edge infrastructure. Building a platform on the edge infrastructure to simplify the implementation of applications is feasible, but imposes another challenge. It is hard to build a platform for edge computing that works efficiently for all application scenarios. Every application scenario has its own characteristics and requirements. If an edge computing platform treats all user applications equally, some applications may not be executed efficiently. For example, suppose a developer has created a user application that writes a data object on the platform. In addition, suppose multiple instances of the user application can be executed simultaneously across the system, while the data object can only be written by one application instance at any time. As a general solution, the platform attaches a write lock to the data object, such that any application intending to write the data object has to acquire its write lock from the cloud before executing the write operation, and release the write lock to the cloud after executing the write operation. Using the cloud as the centralized lock server is necessary because multiple application instances may require the write lock simultaneously from different edge nodes. This solution is effective for all applications, and is even efficient for many of them, but is not efficient for all of them. For example, multiple instances of the user application may be executed simultaneously, but in most of time, they are executed on the same edge node. In such a case, the write lock should be managed on the edge node, rather than in the cloud, which can significantly improve the write performance of the application instances.

In light of this, we propose EdgeEngine, an efficient and customizable framework for edge computing. The main design goal of EdgeEngine is to provide satisfactory performance for user applications with minimum developer effort. To achieve this, we design EdgeEngine as follows. First, in most application scenarios, developers do not need to care about how their applications are executed on the underlying edge infrastructure, e.g., whether their applications are executed on the edge nodes or in the cloud. EdgeEngine automatically manages their applications on the edge infrastructure and achieves satisfactory performance, by applying a set of default policies to the applications. Relying on the default policies, developers could get rid of the details of the underlying edge infrastructure and focus on the implementation of the applications' main logic, which greatly reduces their development workload. Second, in the application scenarios where the default policies cannot guarantee satisfactory performance, as the one described in the previous paragraph, EdgeEngine depends on the developers' knowledge to efficiently manage the applications on the edge infrastructure. More specifically, EdgeEngine provides a flexible programming interface, through which developers could implement user-specified policies on their applications. The user-specified policies are essentially developers' suggestions, which could help EdgeEngine make a better fit to the edge infrastructure in these application scenarios. We carefully design the programming interface, such that developers could implement user-specified policies usually with only several lines of code.

Based on EdgeEngine's design, we implement a prototype system. We further deploy this prototype system on a testbed and evaluate its performance, proving that EdgeEngine could work efficiently in the real world.

3.2 Related Work

In this section, we first give a brief overview on edge computing, and then introduce several widely-adopted distributed computing frameworks from the industry, with which EdgeEngine shares some of the ideas in the design philosophy.

3.2.1 Research on Edge Computing

Edge computing is also termed "fog computing" [14, 122, 120, 121], "mobile-edge computing" [119] and "cloudlets" [86, 85]. These terms contain some subtle differences, but their essential ideas are the same, i.e., providing hardware resources at the edge of network in an IoT or IoT-like environment. Therefore, we will not strictly distinguish these terms in this work.

The idea of cloudlets has been proposed by the academia in the year of 2009 [85]. Later, in 2012, fog computing has been proposed by the industry [14]. After that, a boom of surveys, discussions and pioneer systems on edge computing has emerged in both the academia and the industry [102, 94, 43, 32, 36, 25, 13, 127, 2, 1, 83, 22, 48, 50, 118, 93, 59, 62, 95, 51, 34, 82, 49, 77, 33, 71, 44, 11, 78, 108, 57, 87, 122, 120, 121, 92, 91, 123, 81, 61, 97, 107, 115, 40, 16, 88, 76, 101, 109].

An assistive system based on Google Glass devices and cloudlets is proposed in [40], which is designed for users in cognitive decline. By processing sensed data captured by Google Glass devices on cloudlets, the assistive system performs real-time scene interpretation. A real-time fall detection system based on fog computing is proposed in [16]. The authors investigate a series of new fall detection algorithms and integrate them into the fog computing infrastructure. Experiments with real-word data demonstrate the efficiency of the system.

Foglets is presented in [88], which is a programming infrastructure for fog computing applications. It provides APIs for storing and retrieving application-generated data on the fog nodes, and primitives for communication among application components. By doing this, Foglets enables the placement of application components, data movement among the components, and migration of computation and state according to the mobility pattern of the sensors. MigCEP is presented in [76], which is a placement and migration method for fog infrastructures. This method can ensure application-defined end-to-end latency restrictions, reduce network utilization and improve live migration performance in a fog computing environment. Urgaonkar et al. study the problem of where and when services should be migrated in an edge computing environment [101]. The authors first model the problem as a sequential decision making Markov Decision Problem (MDP), and then solve it in a novel way, i.e., they reduce the MDP to two independent MDPs on disjoint state spaces, and leverage the technique of Lyapunov optimization over renewals to solve the decoupled problems. Wang et al. propose a brand-new fog computing structure, which ensures reliable 3G services for fast-moving users, such as the passengers on a train [109]. They perform a series of theoretical and empirical analyses, demonstrating that their approach could improve the reliability of 3G connections on fast-moving trains.

Our work distinguishes itself from all the previous work, as we try to build a framework for edge computing, which could help developers implement efficient edge computing applications, while requires no or little effort from developers into dealing with the details of the underlying edge infrastructure.

3.2.2 Distributed Computing Frameworks

There are many open-source distributed computing frameworks that share part of the design philosophy with Edge-Engine, such as Apache Hadoop [9], Apache Storm [8] and Apache Spark [7].

Apache Hadoop is a distributed computing framework for big data storage and processing. The core of Apache Hadoop contains two parts: a storage part called HDFS (i.e., Hadoop Distributed File System) and a processing part that implements the MapReduce programming model [31]. Apache Hadoop can achieve good processing performance by batching the requests at the MapReduce part. Nevertheless, the storage part may introduce much overhead to the framework, since HDFS uses secondary storage devices for data storage and may need to apply complicate data manipulation to the dataset before dispatching it to the worker nodes. As a result, Apache Hadoop cannot provide satisfactory performance in real-time big data processing scenarios. In contrast, Apache Storm and Apache Spark are good candidates for such scenarios. Both of them maintain data in main memory and thus eliminate the overhead introduced by disk I/O. Apache Storm implements the stream processing model and dispatches data to the worker nodes for data processing, while Apache Spark batches computational requests and dispatches them to the location where the input datasets reside. Note that Apache Spark can also work in the streaming mode just as Apache Storm.

EdgeEngine shares some of the design philosophy with these distributed computing frameworks. For example, EdgeEngine also implements the stream processing model, i.e., computation is triggered by data input. Furthermore, both EdgeEngine and the three distributed computing frameworks describe their tasks as directed acyclic graphs (e.g., EdgeEngine workflows, Apache Storm topologies, etc.). Nevertheless, different from the three distributed computing frameworks that are designed for big data processing in general, EdgeEngine is particularly designed for data processing in the context of edge computing. In other words, EdgeEngine provides many features that are beneficial for edge applications, and to the best of our knowledge, none of the features are provided by existing distributed computing frameworks.

3.3 EdgeEngine Overview

In this section, we elaborate on the design of EdgeEngine. We first introduce the edge computing infrastructure upon which EdgeEngine is built, and then go deep into the design details of EdgeEngine, elaborating its concepts and discussing how the platform and the programming interface work as a whole, with several concrete examples.

3.3.1 Hardware Infrastructure

Figure 3.1 illustrates the edge computing infrastructure upon which EdgeEngine is deployed. There is a cloud residing at the core of network, while edge nodes distribute at the edge of network. Client devices connect to edge nodes through wireless links. We call the client devices, the edge nodes and the cloud *entiti*es in EdgeEngine.

3.3.2 Workflow Examples

In EdgeEngine, users first define *workflow*s and register them to the system. Afterwards, they could initiate *task*s which are essentially workflow instances. Each workflow contains

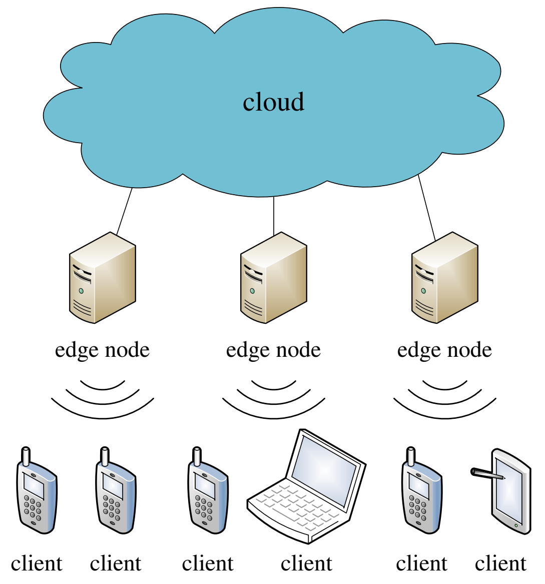


Figure 3.1: EdgeEngine hardware infrastructure.

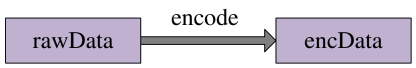
one or more *data item*s and zero or more *transition*s. Transitions consume *input* data items and produce *output* data items.

A data item is either *initial* or *derived*. If a data item is generated by a transition, it is derived. Otherwise it is initial, meaning that users directly send data to this data item in order to initiate tasks. Moreover, a data item is either *durable* or *temporary*. A durable data item is part of the final result of the task. In contrast, a temporary data item is an intermediate result of the task. Therefore, after a task has been finished, its durable data items should be permanently saved and eventually synchronized to the cloud, while its temporary data items could be eliminated at a suitable time.

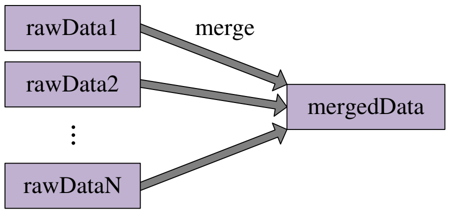
Figure 3.2(a) illustrates a simple workflow, which is called RawVideo. This workflow has only one data item, depicted "rawData" in the figure, and no transition. Clearly, the only data item rawData represents the raw video data that the EdgeEngine system has received from the client device. This data item is both initial and durable.

Figure 3.2(b) illustrates a slightly complicated workflow, which is called EncodedVideo. This workflow contains two data items, rawData and encodedData, and one transition,

(a) The RawVideo workflow.



(b) The EncodedVideo workflow.



(c) The TemperatureDistribution workflow.

Figure 3.2: Workflow examples.

encode. The encode transition takes rawData as input and generates encodedData as output. Clearly, in this workflow, the EdgeEngine system receives raw video data from the client device, encodes it, and saves the encoded data for future use. The rawData data item is initial, and could be either durable or temporary, depending on the application scenarios in which it is involved. The encodedData data item is derived and durable.

Figure 3.2(c) illustrates an EdgeEngine workflow, which is called TemperatureDistribution, that involves multiple writers. Suppose there are N edge nodes covering different regions and connecting to the cloud. Each edge node has a group of temperature sensors distributed around, which continuously send temperature data to it. The edge node in turn forwards the data to the cloud. The cloud receives temperature data from the N edge nodes, merges them, and saves the merged data for future use. The rawData data items are initial, and could be either durable or temporary. The mergedData data item is derived and durable.

3.3.3 System Architecture

Figure 3.3 depicts the architecture of EdgeEngine. There are five layers in the figure. The top layer is called *Application Layer*, where the user applications reside. The user applications initiate workflow instances by writing input data to them, and receive the

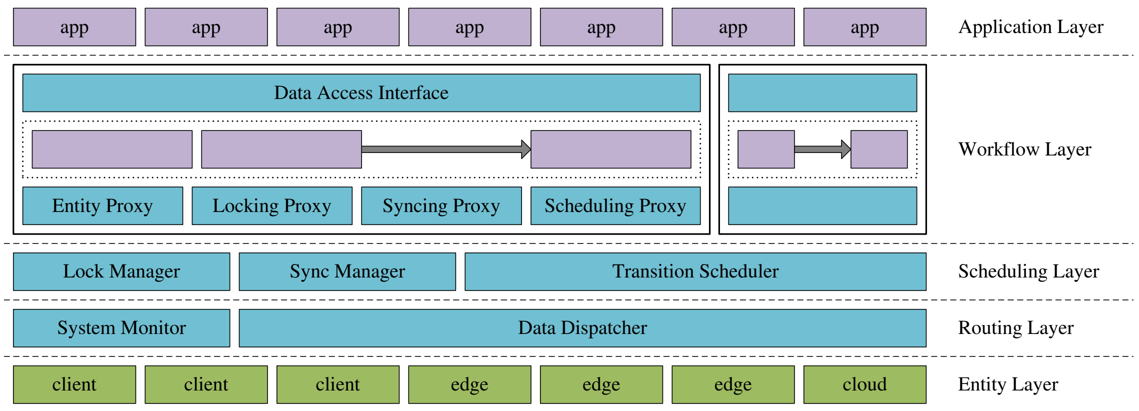


Figure 3.3: EdgeEngine software stack.

results by reading the output data from them. The next layer is called *Workflow Layer*, where the workflow instances reside.

Each workflow instance exposes a *Data Access Interface* to the user applications, through which the user applications read/write its data items. Moreover, each workflow instance has four proxies, i.e., the *Entity Proxy*, the *Locking Proxy*, the *Syncing Proxy* and the *Scheduling Proxy*. These proxies are used to implement user-specified policies on the workflows. Under the Workflow Layer is the *Scheduling Layer*, where three system components, the *Lock Manager*, the *Sync Manager* and the *Transition Scheduler*, reside. These components implement some of the fundamental system mechanisms, i.e., the locking mechanism, the syncing mechanism and the scheduling mechanism. A workflow instance could communicate with these components through its Locking Proxy, Syncing Proxy and Scheduling Proxy, respectively. These components enforce the system default policies as well as user-specified policies on the workflow instances, and therefore are essential in achieving the main design goal of EdgeEngine. Developers could rely on the system default policies enforced by these components in most cases, getting rid of the details of the underlying edge infrastructure, such as whether the workflow instances should be executed on the edge node or in the cloud. They could also provide user-specified policies in the cases where their knowledge of the workflows could help. These

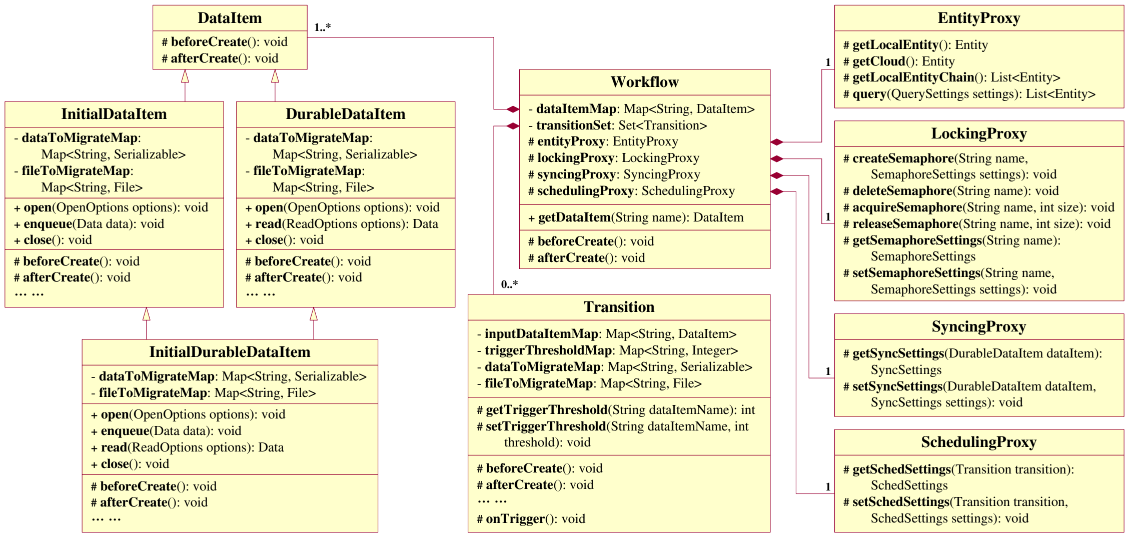

DataItem
# beforeCreate(): void
# afterCreate(): void
1..*
InitialDataItem
- dataToMigrateMap: Map<String, Serializable>
- fileToMigrateMap: Map<String, File>
+ open(OpenOptions options): void
+ enqueue(Data data): void
+ close(): void
# beforeCreate(): void
# afterCreate(): void
… …
DurableDataItem
- dataToMigrateMap: Map<String, Serializable>
- fileToMigrateMap: Map<String, File>
+ open(OpenOptions options): void
+ read(ReadOptions options): Data
+ close(): void
# beforeCreate(): void
# afterCreate(): void
… …
InitialDurableDataItem
- dataToMigrateMap: Map<String, Serializable>
- fileToMigrateMap: Map<String, File>
+ open(OpenOptions options): void
+ enqueue(Data data): void
+ read(ReadOptions options): Data
+ close(): void
# beforeCreate(): void
# afterCreate(): void
… …
Workflow
- dataItemMap: Map<String, DataItem>
- transitionSet: Set<Transition>
# entityProxy: EntityProxy
# lockingProxy: LockingProxy
# syncingProxy: SyncingProxy
# schedulingProxy: SchedulingProxy
+ getDataItem(String name): DataItem
# beforeCreate(): void
# afterCreate(): void
0..*
Transition
- inputDataItemMap: Map<String, DataItem>
- triggerThresholdMap: Map<String, Integer>
- dataToMigrateMap: Map<String, Serializable>
- fileToMigrateMap: Map<String, File>
# getTriggerThreshold(String dataItemName): int
# setTriggerThreshold(String dataItemName, int threshold): void
# beforeCreate(): void
# afterCreate(): void
… …
# onTrigger(): void
EntityProxy
# getLocalEntity(): Entity
# getCloud(): Entity
# getLocalEntityChain(): List<Entity>
# query(QuerySettings settings): List<Entity>
1
LockingProxy
# createSemaphore(String name, SemaphoreSettings settings): void
# deleteSemaphore(String name): void
# acquireSemaphore(String name, int size): void
# releaseSemaphore(String name, int size): void
# getSemaphoreSettings(String name): SemaphoreSettings
# setSemaphoreSettings(String name, SemaphoreSettings settings): void
1
SyncingProxy
# getSyncSettings(DurableDataItem dataItem): SyncSettings
# setSyncSettings(DurableDataItem dataItem, SyncSettings settings): void
1
SchedulingProxy
# getSchedSettings(Transition transition): SchedSettings
# setSchedSettings(Transition transition, SchedSettings settings): void
1

Figure 3.4: EdgeEngine programming interface.

user-specified policies, however, should be implemented with little effort. Therefore, the system components still save the developers from the details of the underlying edge infrastructure in such cases. Below the Scheduling Layer is the *Routing Layer*, where two system components, the *System Monitor* and the *Data Dispatcher*, reside. This layer integrates the entities across the system and provides even more fundamental system mechanisms, e.g., tracking the changes of the entities, monitoring their resource usages and dispatching data among them, etc. A workflow instance could communicate with the System Monitor through its Entity Proxy. The bottom layer is called *Entity Layer*, as the system entities (client devices, edge nodes and the cloud) reside in this layer.

3.3.4 Programming Interface

As mentioned previously, developers could guide EdgeEngine in efficiently utilizing edge nodes through the programming interface. In this section, we describe the design of this programming interface with several examples.

The programming interface is embedded into the Java programming language. Figure 3.4 depicts the *interface class*es and *helper class*es provided by this programming interface. There are six interface classes for defining workflows, i.e., *Workflow*, *DataItem*, *InitialDataItem*, *DurableDataItem*, *InitialDurableDataItem* and *Transition*. As illustrated in the figure, a Workflow contains one or more DataItems and zero or more Transitions. Note that InitialDataItem and DurableDataItem inherit from DataItem, while InitialDurableDataItem inherits from both InitialDataItem and DurableDataItem. This inheritance hierarchy is necessary for defining the Data Access Interfaces and the policies for different types of data items.

Every member variable/function shown in Figure 3.4 starts with either a "+" bullet, a "#" bullet, or a "-" bullet. A "+" bullet indicates a public member, a "#" bullet indicates a protected member, while a "-" bullet indicates a private member. Clearly, the public member functions of the interface classes compose the Data Access Interface of the workflow. User applications could access the data items of the workflow by invoking these member functions.

The protected member functions of the interface classes enclosed in the bottom blocks are *callback* functions. These callback functions are the places where *polici*es on the workflow are defined. As a goal of EdgeEngine is to involve user guidance only when necessary, EdgeEngine implements the default policies in these callback functions for all workflows. Developers could *override* these callback functions to implement their own policies, guiding the system in making a better fit to the underlying edge environment, based on their knowledge on the workflows' application scenarios. To achieve better readability, Figure 3.4 does not show all the callback functions of the interface classes. For example, InitialDataItem has six callback functions beside the two shown in the figure, i.e., *beforeOpen*(), *afterOpen*(), *beforeEnqueue*(), *afterEnqueue*(), *beforeClose*() and *afterClose*(). We use "... ..." to indicate the omission of such callback functions.

When programming these callback functions to implement user-specified policies, developers are supposed to invoke the protected member functions of the helper classes to communicate with the underlying system components. As mentioned, each workflow contains four proxies, which are provided as helper classes in the programming interface, i.e., *EntityProxy*, *LockingProxy*, *SyncingProxy* and *SchedulingProxy*, as depicted in Figure 3.4.

There are three types of policies defined on each workflow, i.e., *locking polici*es, *syncing polici*es and *scheduling polici*es. Locking policies are defined on both initial and durable data items, as initial data items may involve write locks while durable data items may involve read locks. Syncing policies are defined on durable data items, as they are eventually synchronized to the cloud. Scheduling policies are defined on transitions, as they are the computational tasks that need to be scheduled in the system. Generally speaking, developers could override the callback functions of initial and durable data items, invoking the locking proxy to implement user-specified locking policies. Likewise, they could override the callback functions of durable data items, invoking the syncing proxy to implement user-specified syncing policies, as well as override the callback functions of transitions, invoking the scheduling proxy to implement user-specified scheduling policies. The entity proxy could be used as an auxiliary for implementing all the three types of policies, but it is usually invoked in conjunction with the scheduling proxy for implementing scheduling policies on transitions. It is worth mentioning that the two protected member functions of Transition, *getTriggerThreshold*() and *setTriggerThreshold*(), could also be used for implementing scheduling policies on transitions.

Transition also has a special callback function, *onTrigger*(), as depicted in Figure 3.4. This callback function is not used for implementing policies; it is the place where the transition's data processing logic is implemented. Therefore, EdgeEngine does not provide the default implementation of this callback function. Developers have to implement this function for the transitions they define. Beside implementing onTrigger(), a developer is also supposed to specify the *input* data items as well as their *trigger threshold*s when defining a transition. The trigger thresholds indicate the enqueued data size of the input data items for triggering onTrigger() of the transition. More specifically, when each of the input data items has enqueued data with a data size no less than its trigger threshold, the Transition Scheduler will automatically invoke the transition's onTrigger() callback function.

An important design choice of EdgeEngine is how to migrate transitions in an efficient way. Indeed, full VM or container migration always imposes high overhead upon the system [106]. Process migration could be better, but still far from being satisfactory. Many previous studies [85, 23, 27, 39] aim at reducing the migration overhead by minimizing the size of data that needs to be transmitted during a migration, without user intervention. We believe, however, that low migration overhead is of great importance in edge environments, and EdgeEngine could significantly reduce the migration overhead with very little user effort. Our solution is the design of the Transition interface class. The philosophy of this design is similar to that of the user-specified policies: it is the developer who defines the transition has the knowledge of how to efficiently migrate the transition. According to this philosophy, we design Transition in the aforementioned way, i.e., a transaction is triggered by its input data items. Moreover, EdgeEngine guarantees that a transition can only be migrated between two consecutive executions of its onTrigger() callback function; it will never be migrated during the execution of onTrigger(). The developer is also supposed to explicitly specify the data in a transition that needs to be transmitted during a migration. Transition provides two member variables, *dataToMigrateMap* and *fileToMigrateMap*, for this purpose. The developer should put the intermediate results, including both volatile data and files, into these Map objects by the end of onTrigger()'s execution, guiding the system in transmitting only the *necessary* data for the transition's migration. By designing Transition in this way, EdgeEngine could reduce the migration overhead to the minimum, while requires very little user effort, as transitions have to be implemented by some means anyway.

Note that even though the transition migration works in the offline mode (i.e., transitions are terminated before migration) in EdgeEngine, the performance can be as good as or even better than that of the online mode (i.e., transitions are paused, migrated and resumed). The reason lies in the fact that the transition migration works under the guidance of the developer in EdgeEngine. To be more specific, the developer is supposed to specify the data that needs to be migrated, which reduces the data transmission overhead of the migration to the minimum. Moreover, the developer controls the computation granularity of the onTrigger() function, so the transition can be migrated in a suitable time, which is similar to the working strategy of the online mode. It is also worth noting that data items also have computation attached, i.e., their callback functions, and they are migrated with the transitions generating them. Therefore, they also have the dataToMigrateMap and fileToMigrateMap member variables, through which a developer could specify the data that needs to be transmitted during the data item's migration.

3.3.5 Examples of Implementing Workflows

```
public class RawVideo extends Workflow {
    protected DataItem rawData;
    public RawVideo() {
        rawData = new RawData(this);
        dataItemMap.put("rawData", rawData);
    }
}
public class RawData extends InitialDurableDataItem {
    protected int totalSize;
    public RawData(Workflow workflow) {
        super(workflow);
        syncingPolicy = new DummySyncingPolicy();
        totalSize = 0;
        dataToMigrateMap.put("totalSize", totalSize);
    }
    protected void afterEnqueue(Data data) throws Exception {
        super.afterEnqueue(data);
        SyncSettings settings = workflow.syncingProxy.getSyncSettings(this);
        settings.setRegion(totalSize, totalSize + Data.size, Entity.Level.CLOUD,
                SyncSettings.Priority.HIGH);
        totalSize += Data.size;
        dataToMigrateMap.put("totalSize", totalSize);
    }
}
```

Figure 3.5: An example of the RawVideo workflow.

Figure 3.5 demonstrates an implementation example of the RawVideo workflow shown in Figure 3.2(a). Clearly, the developer specifies that the rawData data item inherits the default locking policy from its super class, InitialDurableDataItem, while overrides the afterEnqueue() callback function to implement a user-specified syncing policy. Through the syncing proxy of the workflow, the developer suggests the system to eagerly synchronize the first one tenth data of the rawData data item to the cloud. The ratio of one tenth is an empirical value that the developer considers appropriate for the workflow's application scenarios. Note that the developer can only suggest but not force the system to apply the user-specified policy. If the system considers that it is inappropriate to apply this policy, it will simply reject the developer's suggestion.

```
public class EncodedVideo extends Workflow {
    protected DataItem rawData;
    protected DataItem encodedData;
    protected Transition encode;
    public EncodedVideo() {
        ... ...
        inputDataItemMap.put("rawData", rawData);
        triggerThresholdMap.put("rawData", 4096);
        encode = new Encode(this, inputDataItemMap, triggerThresholdMap);
        transitionSet.add(encode);
    }
}
... ...
public class Encode extends Transition {
    protected MotionVectorBitmap mvBitmap;
    ... ...
    public Encode(Workflow workflow, Map<String, DataItem> inputDataItemMap,
            Map<String, Integer> triggerThresholdMap) {
        super(workflow, inputDataItemMap, triggerThresholdMap);
        schedulingPolicy = new DummySchedulingPolicy();
        mvBitmap = /* empty motion vector bitmap */;
        dataToMigrateMap.put("mvBitmap", mvBitmap);
        ... ...
    }
    protected void afterCreate() throws Exception {
        super.afterCreate();
        SchedSettings settings = workflow.schedulingProxy.getSchedSettings(this);
        settings.setTimeout(5, TimeUnit.SECONDS, SchedSettings.Priority.LOWEST);
    }
    protected void onTrigger() throws Exception {
        ... ...
        Data input = workflow.rawData.read(options);
        /* Generate output by encoding input and update mvBitmap. */
        ... ...
        workflow.encodedData.enqueue(output);
    }
}
```

Figure 3.6: An example of the EncodedVideo workflow.

Figure 3.6 demonstrates an implementation example of the EncodedVideo workflow shown in Figure 3.2(b). The developer implements the encode transition, specifying that whenever rawData has enqueued 4 KB data, the transition be triggered. The developer also implements a user-specified scheduling policy on the encode transition, suggesting the system to eagerly migrate it to the cloud if the transition has not been scheduled on the edge node for more than 5 seconds. The 5 second timeout is an empirical value that the developer considers appropriate for the workflow's application scenarios. The member variable *mvBitmap* is used across the executions of onTrigger(), so the developer specifies that its data should be transmitted during the transition's migration, by putting it into the dataToMigrateMap object.

```
public class TemperatureDistribution extends Workflow {
    protected DataItem rawData1;
    ... ...
    public TemperatureDistribution() {
        /* m: number of sensors belonging to the edge node. */
        rawData1 = new RawData(this, m);
        dataItemMap.put("rawData1", rawData1);
        ... ...
    }
}
public class RawData extends InitialDataItem {
    protected String semName;
    public RawData(Workflow workflow, int semSize) {
        super(workflow);
        lockingPolicy = new DummyLockingPolicy();
        semName = /* a random UUID string */;
        dataToMigrateMap.put("semName", semName);
        SemaphoreSettings settings = new SemaphoreSettings();
        settings.setSize(semSize);
        settings.setAffinity(Entity.Level.EDGE, SemaphoreSettings.Affinity.HIGHEST);
        workflow.lockingProxy.createSemaphore(semName, settings);
    }
    protected void beforeOpen(OpenOptions options) throws Exception {
        super.beforeOpen(options);
        workflow.lockingProxy.acquireSemaphore(semName, 1);
    }
    protected void afterClose() throws Exception {
        super.afterClose();
        workflow.lockingProxy.releaseSemaphore(semName, 1);
    }
}
... ...
```

Figure 3.7: An example of the TemperatureDistribution workflow.

Figure 3.7 demonstrates an implementation example of the TemperatureDistribution workflow shown in Figure 3.2(c). The developer implements a user-specified locking policy on the rawData1 data item, suggesting the system to keep its semaphore on the edge node when the semaphore is being released. EdgeEngine normally releases a semaphore to the cloud for achieving global fairness on locking behaviors, unless it finds that the semaphore is always required by a particular set of entities. To let the system discover that the semaphore on rawData1 is always required by only one edge node may take some time, making the system performance unsatisfactory. As the developer has this knowledge a priori, she could suggest the system to keep the semaphore on the edge node from the very beginning, by using the implementation shown in Figure 3.7. This could help the system avoid the potential performance loss on handling the semaphore.

3.4 Implementation

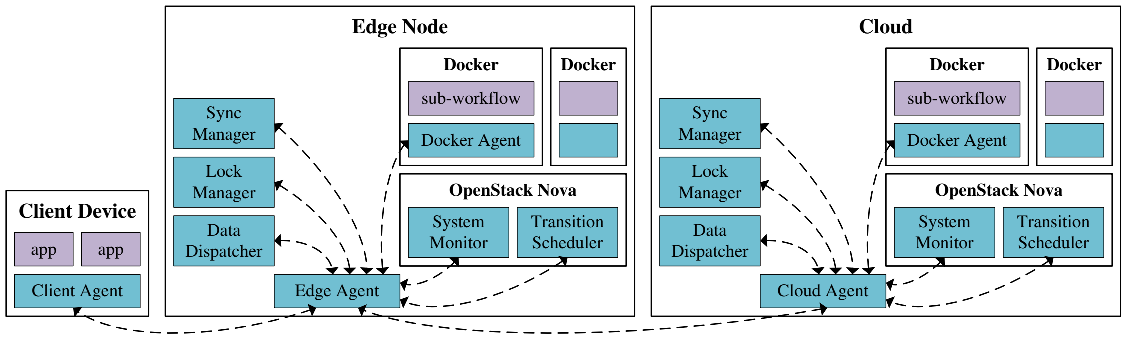


Figure 3.8: EdgeEngine prototype architecture.

To demonstrate EdgeEngine is an efficient framework for edge computing environments, we implement a prototype of it, using Java and Python. Figure 3.8 depicts the architecture of this prototype system.

There are three kinds of entities shown in Figure 3.8: the client device, the edge node and the cloud. We restrict that user applications can only run on client devices while workflow instances can only run on edge nodes and the cloud in our current implementation. Nevertheless, it is also feasible to implement EdgeEngine in the way that both user applications and workflow instances can run on any entity across the system. We can observe from Figure 3.8 that transitions are hosted by Docker containers which are managed by the OpenStack framework. Docker containers provide similar isolated environments as those provided by virtual machines, but are more lightweight and hence suitable for our implementation. OpenStack provides fundamental services for managing the Docker containers, which is essential in providing high performance, so we also employ OpenStack as the underlying platform.

It is worth noting that we install a full OpenStack system on each of the edge nodes and the cloud, rather than install only one OpenStack system to integrate all of them. To integrate these system entities, we implement the *agent* applications. As shown in Figure 3.8, the client device has a *Client Agent*, the edge node has an *Edge Agent*, and the cloud has a *Cloud Agent*. These agents communicate with each other via TCP sockets to enforce system behaviors. As the Docker containers provide isolated environments, we also deploy a *Docker Agent* in each Docker container, which communicates with the Edge Agent for managing the hosted sub-workflow. As mentioned, data items also have computation attached and are migrated with the transitions generating them. In our implementation, a data item is hosted by the same Docker container that hosts the producer transition. (Initial data items are exclusively hosted by Docker containers.) Therefore, we use “sub-workflow” in Figure 3.8 to indicate both the transition and the data items it generates, rather than simply use “transition”.

Data Dispatcher, Lock Manager and Sync Manager shown in Figure 3.3 are implemented as distributed applications which communicate with each other through the agent applications for managing the data items. System Monitor and Transition Scheduler, on the other hand, are integrated into the OpenStack Nova module, as OpenStack is the underlying platform that manages Docker containers and tracks the resource usages.

3.4.1 System Monitor

System Monitor manages the local connectivity information, i.e., the view of all the entities connecting to the local entity. This information is needed by other system components. For example, when a client device switches from one edge node to another, the System Monitors on these two edge nodes will exchange information, guaranteeing that their Data

Dispatchers can work correctly after the edge node switch. Moreover, System Monitor tracks the resource usages if the local entity is an edge node. It relies on the OpenStack Nova module to collect the CPU and memory usages, while independently collects the network usage. The resource usage information is also needed by other system components, such as the Transition Scheduler and the Sync Manager.

3.4.2 Data Dispatcher

Data Dispatcher is responsible for reliably dispatching data between the system entities. It retrieves the local connectivity information from System Monitor, and establishes data channels with all connected entities. It also records the UUIDs of the managed workflow instances, so that it could dispatch data for any workflow instance the local entity manages through the correct data channel. When a data channel is temporarily unavailable, Data Dispatcher will cache the data that should be transmitted through the data channel on the local storage, and transmit the cached data immediately after the data channel has been re-established. During an edge node switch, the Data Dispatchers on the two edge nodes will also communicate with each other to transfer the ownership of the workflow instances that are initiated by the client device.

3.4.3 Lock Manager

Lock Manager implements the fundamental locking mechanism in the prototype system, by managing the semaphores created by both the system default and user-specified locking policies. It uses the cloud as the centralized server to manage the semaphores. As the main goal of EdgeEngine is to guarantee high performance in most cases without user intervention, while adjust its behaviors according to user guidance in the other cases, we implement Lock Manager as follows.

After a semaphore has been created, the Lock Manager on the cloud builds an *affinity mapping* for this semaphore. By default, the cloud's Lock Manager specifies that every edge nodes has the affinity *MEDIUM* on the semaphore. After that, the cloud's Lock Manager tracks the usage of the semaphore. If the semaphore is continuously required by only one edge node, the cloud's Lock Manager will gradually increase the edge node's affinity on this semaphore. The edge node could then hold the semaphore after it has been released for some time, the length of which depends on the edge node's affinity value. If the semaphore is suddenly required by another edge node, the affinity mapping on the semaphore will be reset. A user-specified locking policy could directly modify the affinity mapping of the semaphore through the workflow instance's Locking Proxy, as shown in Figure 3.7.

3.4.4 Sync Manager

Sync Manager implements the fundamental syncing mechanism for durable data items. A well-designed syncing mechanism is essential in guaranteeing both good system performance and both good user experience. In light of this, we implement Sync Manager as follows.

The Sync Manager on each edge node continuously retrieves the network usage from System Monitor on the same edge node. According to the currently available network bandwidth from the edge node to the cloud, the Sync Manager knows how much data could be immediately synchronized. By default, the Sync Manager fairly synchronizes the durable data items based on this knowledge. A user-specified syncing policy, however, could modify the priority of a durable data item through the workflow instance's Syncing Proxy, as shown in Figure 3.5.

3.4.5 Transition Scheduler

Transition Scheduler schedules the transitions for workflow instances on both the edge nodes and the cloud, and migrates them when necessary. Clearly, an efficient implementation of Transition Scheduler is essential in guaranteeing good system performance.

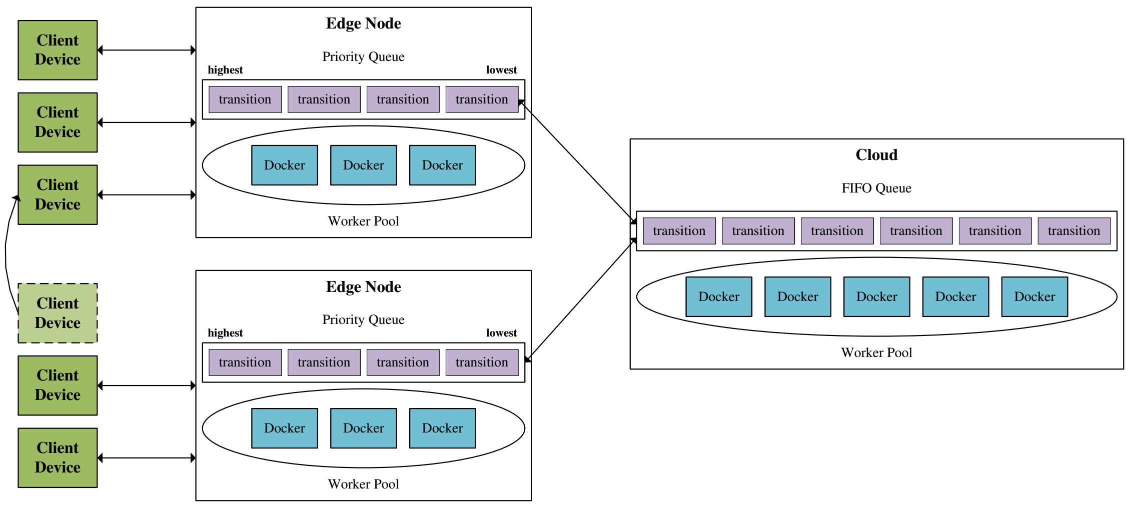


Figure 3.9: Transition scheduling in EdgeEngine.

Figure 3.9 demonstrates how Transition Scheduler works in EdgeEngine. Note that the transitions shown in the figure are actually sub-workflows described earlier, because the output data items are migrated with the transitions that generate them. Nevertheless, we still use “transition” in the figure to simplify our explanation.

From Figure 3.9, we can observe that each edge node as well as the cloud maintains a *worker pool*. The number of Docker containers in each worker pool is fixed. In other words, the Transition Scheduler on each edge node as well as that in the cloud only allows a fixed number of Docker containers to execute simultaneously. Meanwhile, the Transition Scheduler on each edge node maintains a priority queue for all the local transitions, and it sorts the transitions in this queue as follows. It records the time length that a transition has been scheduled, as well as the size of the input data the transition has processed. If a transition has processed a lot of input data in a short time, it is assigned a high priority. In contrast, if a transition has processed only a small amount of data in quite a long time, it is assigned a low priority. The queue is then sorted by the priorities of the transitions, from the highest to the lowest. When the Transition Scheduler decides to schedule a suspended transition, it picks up the first transition from the head of the priority queue and sends it to an available Docker container. A new transition is assigned a high priority for more chance to be scheduled. If a transition has not been scheduled for some time, the Transition Scheduler will also decrease its priority.

The Transition Scheduler on each edge node also imposes a restriction on the priority queue's length. If the priority queue is longer than two times of the maximum number of Docker containers, the Transition Scheduler will pick up transitions from the tail of the priority queue and migrate them to the cloud. On the other hand, if the priority queue is shorter than the maximum number of Docker containers, the Transition Scheduler will try to migrate transitions from the cloud. The factor values of two and one are empirical. In contrast, the Transition Scheduler in the cloud maintains a FIFO queue rather than a priority queue, and it imposes no restriction on the FIFO queue's length. Nevertheless, it also calculates the priority of each transition in the background, and will migrate transitions of high priority to the edge nodes when possible.

By implementing Transition Scheduler in the way described above, EdgeEngine could efficiently utilize the resources on the edge nodes. This is because a transition of a high priority implies that it will probably not consume much computational resource on the edge node, but will consume a lot of network resource when being migrated to the cloud. Therefore, it will be beneficial to execute the transition on the edge node rather than in the cloud.

Another design choice that must be made when implementing such a Transition Scheduler is that every client device is mobile and may switch the the edge node it connects to during the execution of a transition, or may be temporarily disconnected from any edge node (but connect to the cloud), and EdgeEngine should be able to provide seamless service in such a scenario. We make this design choice in a simple way in our implementation. When the client device is disconnected from the current edge node, EdgeEngine will unconditionally migrate the transition to the cloud. When the client device connects to a new edge node, EdgeEngine will migrate the transition from the cloud to the new edge node if the priority queue on the new edge node is shorter than two times of the maximum number of Docker containers. More complicated scheduling algorithms and programming interface support for this design choice are left as our future work.

3.5 Evaluation

We deploy our prototype system on a testbed and conduct experiments on it. The testbed consists of five servers, one of which is more powerful than the others. The more powerful server is used as the cloud server, while the others are used as edge nodes. The cloud server has an 8-core Intel i7 CPU with a clock speed of 4.00 GHz and 16 GB main memory. Each edge node has a 4-core CPU with a clock speed of 2.83 GHz and 4 GB main memory. The edge nodes are directly connected to the cloud server through a 1000 Mbps network link. To simulate a real-world edge computing environment, we set the upper bound of the network bandwidth between the edge nodes and the cloud server to 40 Mbps, and the latency to 10 ms (i.e., the round trip time is 20 ms). For simplicity of system deployment, we do not employ real client devices for the experiments, but deploy an application on each edge node to simulate the behaviors of the connected client devices. The simulated wireless network between a client device and the edge node it connects to has a maximum bandwidth of 8 Mbps, and the latency is negligible.

3.5.1 Evaluation on the Programming Interface

Table 3.1: Efficiency of the programming interface.

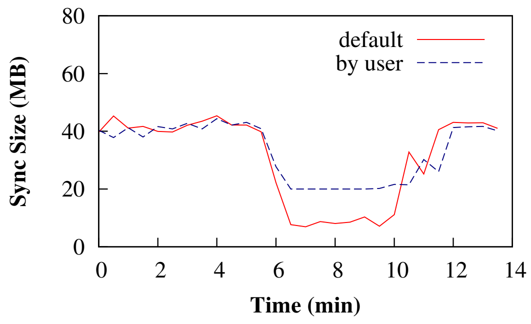
	LoC on EdgeEngine	**LoC from Scratch**
Syncing Policy	9	> 400
Scheduling Policy	5	> 600
Locking Policy	14	≈ 100

As mentioned, a design goal of the EdgeEngine programming interface is to minimize the effort a developer must put into implementing her own policies when necessary. In light of this, we evaluate the efficiency of this programming interface. More specifically, we compare the lines of code (LoC) that are required to implement the user-specified policies shown in Figure 3.5, 3.6 and 3.7, through the EdgeEngine programming interface and from scratch, respectively. Table 3.1 shows the results. Note that implementing a user-specified policy from scratch is not equivalent to only implementing the developer's guidance. For example, to migrate the encode transition shown in Figure 3.6 according to the developer's guidance is simple, but the user-specified scheduling policy is more than that. It is a combination of the system's default scheduling policy and the developer's guidance. Consider that the transition may also be migrated back to the edge node when possible, as described in Section 3.4.5. Therefore, we estimate that the user-specified scheduling policy requires more than 600 LoC from scratch, by checking our implementation of the Transition Scheduler component. Clearly, the EdgeEngine programming interface could greatly reduce the workload of implementing the user-specified policies, as it usaually requires only several LoC.

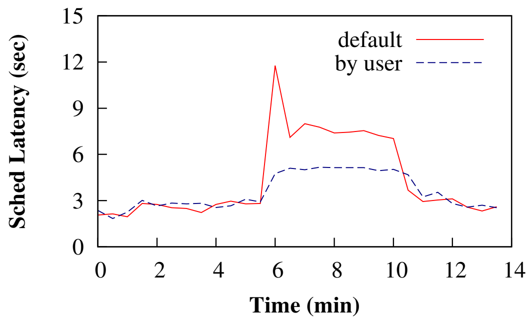
3.5.2 Evaluation on the System Components

The main design goal of EdgeEngine is to guarantee high performance both with and without user guidance. The system components shown in Figure 3.3 are critical in achieving this goal, as they are responsible for enforcing the system default as well as user-specified policies. We evaluate their performance in this section. Figure 3.10 illustrates the results.

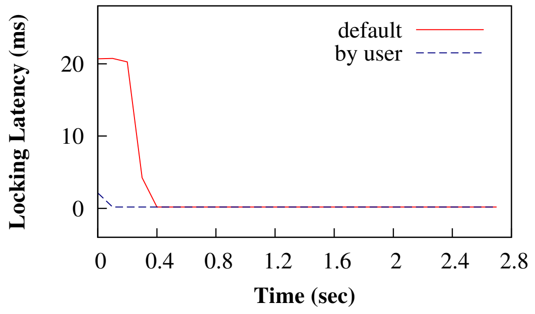
Figure 3.10(a) shows the performance of the Sync Manager component when enforcing the default syncing policy as well as the user-specified syncing policy shown in Figure 3.5. In this experiment, we simulate the scenario in which 200 MB RawVideo workflow instances arrive at our prototype system. More specifically, they arrive at each edge node independently, with the arrival intervals following an $N(10$ sec, 4 sec$^2)$ distribution in the first 6 minutes, then following an $N(1$ sec, 0.25 sec$^2)$ distribution in the next 4 minutes, and finally following an $N(10$ sec, 4 sec$^2)$ distribution in the last 4 minutes. Figure 3.5 shows the average size of data that has been synchronized for each workflow instance.



(a) Performance on the syncing policies.



(b) Performance on the scheduling policies.



(c) Performance on the locking policies.

Figure 3.10: Performance of the system components.

Clearly, Sync Manager synchronizes less data under the default syncing policy than under the user-specified policy when the system is overloaded, because the default policy only guarantees that the first one tenth of data is eagerly synchronized for each workflow instance, while the user-specified policy suggests Sync Manager to eagerly synchronize all data for the workflow instances. However, the default policy also performs as good as the user-specified policy when the system is not overloaded. When the system is overloaded, the default policy tries to synchronize as much data as possible, without causing any potential side effect, such as increasing the user-perceived response time. (Note that if Sync Manager considers that the user-specified policy may cause a serious side effect, it will reject the user-specified policy.) Moreover, the default policy does not require any effort from developers, in contrast to the user-specified policy, which still requires several LoC. We therefore expect that developers only implement user-specified syncing policies when necessary, while directly employ the default syncing policy in most cases.

Figure 3.10(b) shows the performance of Transition Scheduler when enforcing the default scheduling policy as well as the user-specified scheduling policy shown in Figure 3.6. We also simulate a changing workload in this experiment. Figure 3.10(b) shows the average latency before the encode transitions get scheduled. Clearly, when the system is overloaded, the transitions have a shorter latency before being scheduled under the user-specified policy. Nevertheless, as shown in the figure, the default policy can quickly adjust the scheduling of transitions according to the system's workload. By doing this, it could also achieve satisfactory performance under high workload, without causing any potential side effect.

Figure 3.10(c) shows the performance of Lock Manager when enforcing the default locking policy and the user-specified locking policy shown in Figure 3.7. We simulate a stable workload on only one edge node in this experiment. Figure 3.10(c) shows the average latency perceived by user applications when requiring the semaphore through the edge node. Clearly, with the developer's knowledge, the user-specified policy could achieve low locking latency from the very beginning. However, as shown in the figure, the default policy can quickly adjust the affinity mapping of the semaphore according to the distribution of user requests. By doing this, it achieves as low locking latency as the user-specified policy achieves (less than 0.5 seconds), exhibiting satisfactory performance.

3.6 Conclusions

In this work, we present EdgeEngine, an efficient and customizable framework for edge computing. EdgeEngine achieves good performance for most application scenarios, requiring no user effort. For the application scenarios where user guidance is necessary for achieving good performance, EdgeEngine provides a programming interface through which users could implement user-specified policies with very little effort, guiding the system in acting efficiently. Evaluation on our prototype system proves that EdgeEngine is practical in the real world.

Chapter 4

Nomad: An Efficient Consensus Approach for Latency-Sensitive Edge-Cloud Applications

4.1 Introduction

Edge computing, also known as cloudlets [85], fog computing [14] and mobile-edge computing [46], is a new paradigm of distributed computing. The basic idea of edge computing is to provide elastic resources at the edge of network, serving the end devices with lower network latency than cloud computing. Since its birth, edge computing has drawn plenty of attention from the industry, because it is able to fulfill the requirements on network latency imposed by many distributed applications. More notably, edge computing is viewed as the enabler of a spectrum of emerging applications, such as IoT applications, big data analytics, and real-time mobile-edge applications.

Many of such applications are large-scale, geo-distributed ones. There could be dozens of separate edge networks in the system, backed by several interconnected cloud data centers. Massive end devices may simultaneously connect to the system via edge servers located at different places. Information is rapidly exchanged between the end devices and the edge servers, and some of the messages sent to the edge need to be spread across the entire system. In fact, many large-scale edge applications build their services atop the functionality that the system orders the messages received by the edge in a timely manner. However, implementing such a functionality is challenging. This is not only because the system may receive messages at a prohibitively high rate, but there may also be a good number of parties involved in, including both the cloud data centers and the edge servers in the edge networks. How to make a consistent decision in a distributed system is a classic problem in the distributed computing area, which is known as the consensus problem. This problem has been studied for decades and is still an attractive topic in the academia.

Among the existing consensus approaches in the literature, the Paxos-based ones ensure strong consistency, i.e., conflicting states will never occur in the system, and they complete the decision-making process as soon as all the conditions have been met, effectively reducing the user-perceived latency. Because we aim at achieving fast ordering on the messages for geo-distributed edge-cloud applications, and many of such applications cannot tolerate inconsistency on the message order, choosing a proper Paxos-based approach seems to be a plausible solution. However, existing Paxos-based approaches have a severe drawback that they fail to support large-scale distributed systems, because the message complexity grows dramatically with the increase of the number of system nodes. As mentioned previously, the applications we study may run on a great number of distributed parties, so the Paxos-based approaches cannot be directly utilized to solve this problem.

To this end, we propose Nomad, a consensus approach that achieves fast message ordering for geo-distributed edge-cloud applications. The main idea of Nomad is to divide the system into two levels, the cloud level and the edge level, and at each level, Nomad runs a consensus protocol that fits the network traits of that level. The two protocols also cooperate to adapt to and take advantage of the workload change in the system. To summarize, the contributions of this work are fourfold.

- We formulate a problem of achieving fast message ordering for geo-distributed delay-sensitive edge-cloud applications, and propose two realistic application scenarios to show the significance of studying this problem.
- We design a novel Paxos-based consensus protocol for the edge level, which rapidly orders the messages in individual edge networks. It dynamically distributes the leadership of a sequence of Paxos instances among the edge servers, based on the recent running history, and introduces a cloud-based arbitrator to quickly resolve the contentions on the edge.
- We design a consensus protocol for the cloud level, which works with the edge-level protocol as a whole. It adopts a lease-based method to opportunistically transfer the control from the cloud data centers to the most heavily-loaded edge network, based on the recent running history.
- We implement a prototype of Nomad, and evaluate it on a testbed. The results show the high efficiency of Nomad. In particular, the edge-level protocol outperforms the existing Paxos-based solutions, such as Multi-Paxos [54], Mencius [63] and E-Paxos [69], under different experimental settings.

4.2 Background

Before elaborating the design of Nomad, we first introduce some preliminaries about edge computing and consensus.

4.2.1 Edge Computing

Edge computing has been proposed as an extension of cloud computing [84]. Its goal is to serve end users at the edge of network, providing better network conditions such as low network latency and high network bandwidth. It has attracted a lot of research effort in recent years [85, 14, 13, 46, 92, 91, 122, 120, 121].

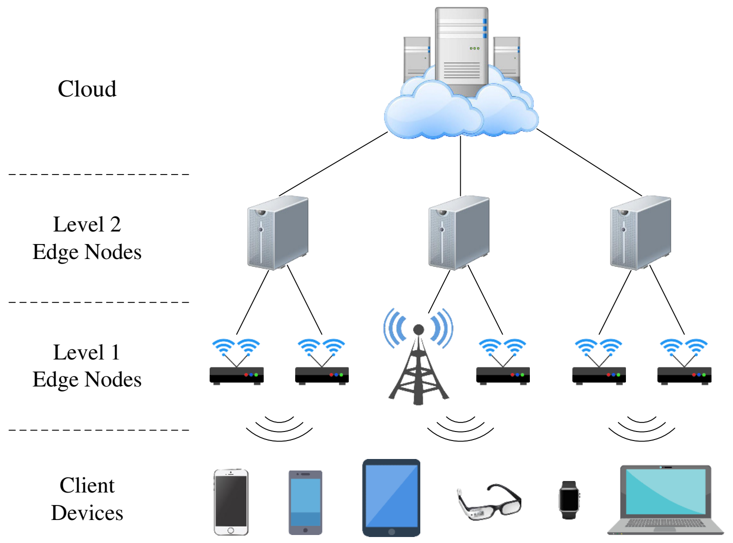


Figure 4.1: A hierarchical edge computing architecture.

Figure 4.1 shows a hierarchical edge computing architecture. Client devices, including wearables, smartphones, tablets and laptops, are wirelessly connected to the level 1 edge nodes. Level 1 edge nodes are usually wireless access points and cellular base stations, which are further connected to the level 2 edge nodes via wired links. There could also be a backend cloud at the core of network. Note that there may be more than two levels of edge nodes in some architectures. Furthermore, some edge computing architectures may only have client devices and edge nodes without having a backend cloud. An edge computing system that possesses one or more backend clouds can also be called an edge-cloud system, and the applications running on it can be called edge-cloud applications.

4.2.2 Consensus

Consensus studies the problem of how to achieve overall system reliability in the presence of a number of faulty nodes in a distributed system. It has been studied for decades but remains a hot research topic in the academia [35, 73, 37, 53, 54, 56, 55, 12, 63, 69, 60, 47, 74, 28, 124, 17]. Among the existing consensus solutions in the literature, the Paxos-

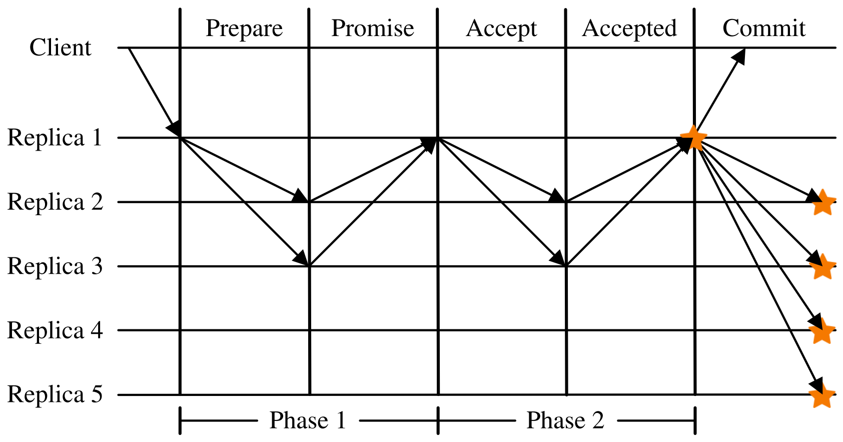


Figure 4.2: The main process of the Paxos algorithm.

based ones are widely adopted by the industry [15, 18], because they guarantee strong consistency during execution, and can work efficiently in many settings.

Figure 4.2 depicts the main process of the Paxos algorithm. There are essentially two phases in Paxos, i.e., Phase 1 and Phase 2. In Phase 1, the replica that has received a value from the client sends Prepare messages to the other replicas, and the other replicas will send back a Promise message if they have not accepted any value. After the initial replica has received the Promise messages from a quorum, i.e., a majority of the replicas, Phase 1 succeeds, and the replica will start Phase 2 by sending Accept messages to the others. Similarly, the other replicas will send back a Accepted message if they haven not accepted any value. After the initial replica has received the Accepted messages from a quorum, the value is acknowledged by the system, and the replica will commit it locally as well as informing the others to commit it. In essence, Phase 1 is for electing the leader, while Phase 2 is for making the system accept a value.

Some Paxos-based solutions in the literature, such as Multi-Paxos [54], Mencius [63] and E-Paxos [69], work in the way that they run a sequence of Paxos instances, and therefore they can be used to determine a unique order on the values received across the system. Multi-Paxos runs the sequence of Paxos instances with a fixed leader, skipping Phase 1 for all Paxos instances, which greatly reduces the communication costs for achieving consensus. However, the fixed leader will become the bottleneck of the system. Mencius eliminates this bottleneck by distributing the leadership evenly among the replicas in a round-robin way. Nevertheless, a slow replica in the system may greatly affect the performance of Mencius. E-Paxos determines the dependencies on the Paxos instances, working around this slow-replica problem. It also improves the system performance by delaying the resolution of conflicts. However, it imposes more communication costs, which may lead to worse performance than Mencius in many cases.

4.3 Motivational Scenarios

To better explain the design of Nomad, we first describe the following two application scenarios as examples.

4.3.1 Internet of Things (IoT) Payments

The Internet of Things (IoT) has grown rapidly in recent years [45]. The concept of IoT is to manage a massive number of smart devices, including RFID tags, sensors, actuators, mobile devices and wearables, with a global network infrastructure. Edge computing is usually considered as the best enabler of IoT systems, because IoT devices possesses limited hardware resources, and edge computing can serve them with low network latency [14, 13, 91]. As IoT devices become more and more ubiquitous in our daily life, making digital payments through IoT devices is being considered as an appealing application [89, 105]. With IoT payments, customers can make purchases anywhere at any time, as long as their client devices, such as smartphones and wearables, are wirelessly connected to a nearby IoT device. This greatly facilitates the purchase process on goods and services and is possible to substantially increase the income of the merchants.

Figure 4.3 illustrates the edge computing infrastructure of the IoT payment application in a global range. Big black circles in the figure are cloud data centers, while small yellow circles are edge nodes. Ellipses illustrate the “backing” range of the clouds; edge nodes

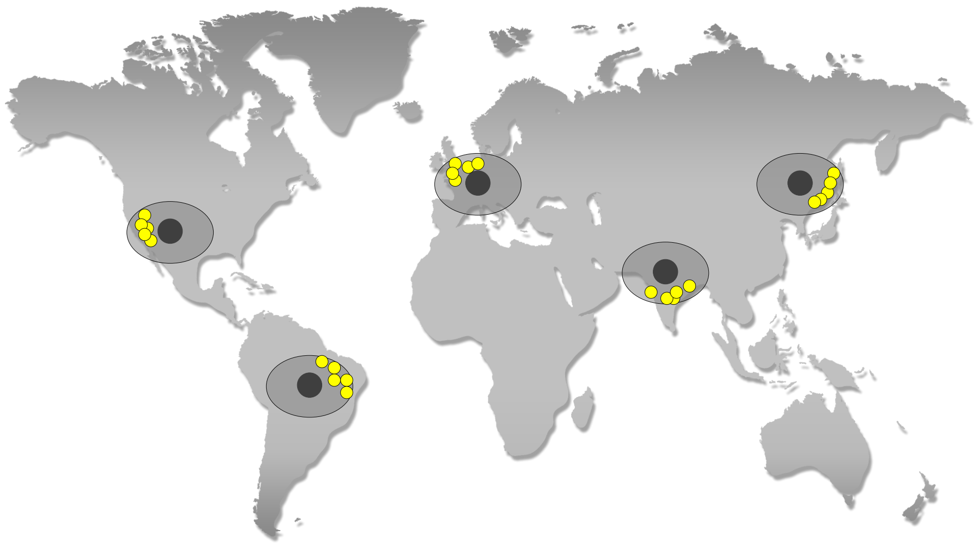


Figure 4.3: The edge computing infrastructure of the IoT payment application.

falling into an ellipse are backed by the cloud at the center. IoT devices receive payment requests from customers, and forward them to the nearest edge node. By this means, payment requests are received by the edge computing infrastructure, and a transaction is triggered by each payment request. All payment transactions need to be globally ordered, in order to guarantee the validity of those transactions.

Notably, the payment workload can be quite unbalanced in such a scenario. The reason is that the payments are usually, if not always, made during daytime, or even only during several time periods, such as noon (lunch break) and evening (off work). Therefore, it is likely that in most of the time, only one cloud region receives very heavy workload, while the others receive very light or even no workload.

4.3.2 Location-based Massively Multiplayer Online Games

Cloud-based online games haven been developed for years, and are still a hot topic in the gaming industry [66, 72]. Edge computing can undoubtedly help improve the user experience of cloud gaming, as it provides lower network delay and higher network bandwidth.

Consider the scenario of a location-based massively multiplayer online game. A game company is planning to launch an online augmented reality (AR) game in the US. Players connect their AR devices, such as smartphones and AR glasses, to the gaming service. The game is location-based, meaning that the map in the game is generated from the real world, and any player's character is at the corresponding location of the player's real world location. When the player is moving, her character is also moving in the game. When multiple players meet each other, their characters in the game also meet each other. An player can fight with other players, cooperate with others to hunt virtual monsters, pick up virtual items on the road, and so on. The game is separated by cities, and most operations of the players are done inside cities they are residing in. Inter-city communications do exist, but are rare. For example, a player in one city may interact with another city only when she has mastered and is casting some special skill such as "teleport".

The game is designed to be capable of supporting a massive number of simultaneously online players in each city. The game is designed to support a massive number of simultaneously online players in each city. As the number of such players can be so huge, the game software is designed to run on the players' client devices, which performs the heavy computations such as video processing locally. As the number of such players can be very large, the game software is designed to run on the players' client devices, performing heavy computations such as video processing. This is reasonable considering the development of the hardware on the client devices in recent years [98]. The computations that the game software performs are deterministic, meaning that the game software always yields the same output given that it starts from the same initial state and is fed the same input. The computations are deterministic, i.e., the game software always yields the same output given that it starts from the same initial state and is fed the same input.

For each city, all the relevant operations from the players, regardless of inside the city (e.g., hunting monsters) or outside the city (e.g., teleporting to the city), should be ordered and fed into the relevant players' client devices. Only in this way, can the consistency on

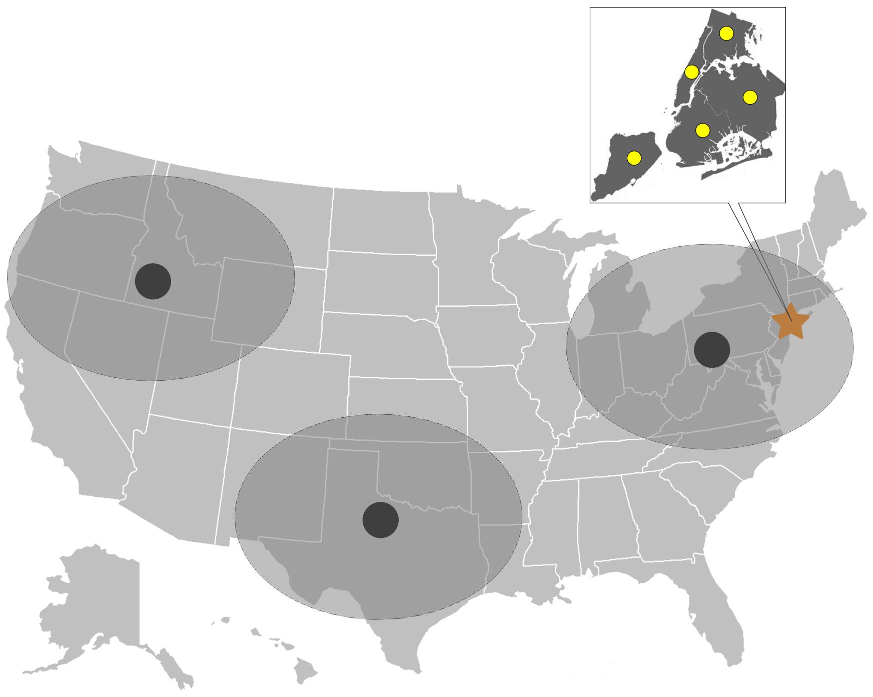


Figure 4.4: The edge computing infrastructure for a location-based multiplayer online game in the US. The New York City is shown at the top right part.

the game's logic for all the players be guaranteed. For this reason, the game company is supposed to build the gaming service for each city as an operation-ordering service. The service can be built completely on a cloud basis, but in order to provide satisfactory user experience under the potentially very high operation rate, relying on edge computing could be a good choice.

Figure 4.4 depicts the edge computing infrastructure of such a location-based multiplayer online game in the US. Similarly, big black circles are cloud data centers, and ellipses are the "backing" range of the clouds at the center. Suppose the three clouds shown in the figure cover all the cities in which the operation-ordering service is provided. As an example, one such city, namely the New York City, is enlarged and shown at the top right part of Figure 4.4. Five edge nodes, represented by small yellow circles in the figure, are deployed in the city. Clearly, the operation-ordering service for any city, such as the New York City shown in Figure 4.4, always receives highly unbalanced workload, because intra-city operations are overwhelmingly more than inter-city operations.

4.4 The Design of Nomad

In this section, we first formulate the problem using the two application scenarios given in Section 4.3, and summarize the assumptions made by our solution. After that, we describe the design of Nomad in detail.

4.4.1 Problem Formulation

For both application scenarios described in Section 4.3, we observe that the edge computing infrastructure, including both the cloud data centers and the edge nodes, is used to order the data received across the system. More specifically, all the data is initially received by the edge nodes, and then spread into the system for ordering. The workload of the applications possesses the following features.

- Every piece of data being ordered is of a small data amount. It is reasonable to consider that the payment transactions and the operations are always less than 1 KB, and in most cases, only tens of bytes.
- The workload can be very heavy. Take the IoT payment application for example. It should be designed to support high transaction rates, as it aims to serve many customers in the global range. Similarly, the location-based game should be able to support a large number of simultaneously online players, who generate operations frequently and rapidly.
- The workload can be highly unbalanced. For example, the IoT payment application serves customers at different time zones, but it is likely that most payments are made during several fixed time periods. In the location-based online game case, intra-city operations are far more frequent than inter-city operations.
- The workload may change dramatically from time to time. Notably, the workload of the IoT payment application may be sometimes concentrated in one region and later

switches to another, leading to changing workload unevenness. On the other hand, the workload of the location-based online game does not change so dramatically, but the workload distribution on the edge nodes may still change from time to time, e.g., during the daytime, many players appear in the office area, while after evening, most of they appear in the residential area.

For both applications, it can also be observed that the user-perceived latency is the most significant indicator of achieving satisfactory user experience. For the IoT payment application, the customers usually make payments while they are walking or driving. If the payments cannot be acknowledged in a short time, the benefits of using the application will be undermined. Similarly, for the location-based game, if a player performs an operation, such as attacking a monster, she would expect to see the outcome as quickly as possible. If the outcome is generated after a long time, the gaming experience will be greatly damaged. If this frequently happens, the game will be soon deserted by the players. In fact, edge computing has been proposed as an extension of cloud computing mainly because it provides lower latency. For this reason, the goal of employing edge computing in most cases, from our understanding, is to achieve good user-perceived latency for the application.

To summarize, the problem we study is how to achieve fast event ordering in an edge-cloud computing environment, under the conditions that 1) the events being ordered are all of small data amounts, and 2) the workload can be very heavy, highly unbalanced, and ever-changing at both the cloud level and the edge level. Note that we employ the term “events” to indicate the data being ordered, rather than using any other word such as “transactions” or “operations”.

4.4.2 Assumptions

To design an effective solution to the problem, we make the following assumptions on the edge-cloud computing system.

- At any time, at most a minority of the cloud data centers may become unreachable, i.e., experiencing failures or network-partitions.
- A cloud data center may back many edge networks that are geographically separated.
- At any time, at most a minority of the edge nodes in the same edge network may become unreachable, i.e., experiencing failures or network-partitions.
- At any time, for any edge network, at least one living cloud can access all the living edge nodes in that edge network.
- The round-trip time (RTT) in an edge network may be uneven, i.e., some edge nodes in the edge network may have a longer network delay when communicating with the other.

These assumptions are reasonable from our point of view. The first and the third assumption are similar to those made by other distributed computing solutions [53, 54, 63]. The second one is derived from the application scenarios described in Section 4.3. The fifth one is derived from the fact that the edge nodes in an edge network may be geographically scattered and have different network distances to the others. The forth one, however, needs more inspections. We make this assumption for the failover purpose. It is possible in the real world that a network partition makes all the edge nodes in an edge network unreachable to the outside. However, as failover is important in many cases, service providers, such as those described in Section 4.3, have the motivation to fulfill this assumption. They may employ some engineering methods, such as setting a backup satellite network in the system, to work around the full network partitions of the edge networks.

4.4.3 Consensus on the Edge

According to the discussion in Section 4.4.1, it is clear that the problem we try to solve is non-trivial. First of all, if the events being ordered are of large data amounts, then the data transfer time will dominate the user-perceived latency. In such a case, whether the ordering solution is a trivial one or a non-trivial one does not make much difference. However, because the problem states that the events are all small data pieces, a well-designed solution can significantly outperform the trivial solutions, such as making an all-to-all communication between all the parties. Second, only using the clouds to order the events is also a plausible solution, but doing so cannot take advantage of edge computing. The high network latency between the end devices and the cloud will undoubtedly deteriorate the performance of the event ordering. Last, using physical timestamps to order the events does not work, because the events should be ordered by the logical time [52] rather than the physical time to guarantee the correct order of the events.

Given that the workload across the system can be highly unbalanced and sometimes concentrates at only one edge network, it could be beneficial to opportunistically order the events at the highly-loaded edge networks, and only involve the remote clouds when necessary. For this reason, we first suppose that the system only consists of one edge network and one backend cloud, and discuss how to achieve fast event ordering in such a situation. A general-case discussion will also be given in the following part of the thesis.

When the system contains only one edge network and one backend cloud, it falls back to the typical form of distributed system. There are several Paxos-based consensus solutions in the literature that can effectively order the data received by a distributed system, such as Multi-Paxos, Mencius and E-Paxos. Their goals, however, are to achieve low latency when the workload is light and high throughput when the workload is heavy. The goal of our solution, in contrast, is achieve as low user-perceived latency as possible in any cases, especially when the workload is heavy.

As such, we design a new Paxos-based consensus protocol for ordering the events on the system containing only one edge network and one backend cloud. Similar to the existing Paxos-based ordering protocols, our protocol also executes a sequence of Paxos instances among the system nodes. More specifically, the Paxos instances are pre-assigned to the edge nodes, i.e., when they are executed, they start from Phase 2 and Phase 1 is considered already finished. As mentioned in Section 4.2.2, Phase 1 is for electing the leader while Phase 2 is for proposing a value. Therefore, the leader nodes of the Paxos instances are artificially determined in advance in our consensus protocol, which reduces the communication cost and thus improves the ordering performance. This does not violate the correctness of consensus, as proved by Mencius. Unlike Mencius, which distributes the leadership of the Paxos instances among the system nodes in a fixed round-robin way, however, our protocol distributes the leadership dynamically according to its running history, for achieving as low ordering latency as possible. The design philosophy of our consensus protocol is summarized as follows.

- To adapt to the workload change on the edge, our protocol dynamically distributes the leadership of the Paxos instances on the edge nodes according to its running history, assigning more leadership to more heavily-loaded edge nodes.

  The intuition of doing so is that the workload has temporal and spatial locality when it is changing, which can be used to predict the workload condition in the near future.

- Any edge node can proactively skip a Paxos instance that belongs to another if no event has been committed in this Paxos instance.

  The intuition of doing so is to guarantee low latency in the presence of differences between the predicted workload and the real workload.

- If an edge intends to commit an event but fails for many times because its Paxos instance has been skipped by the others, it delegates the event to the backend cloud.

  The intuition of doing so is to reduce the side effects caused by the skipping scheme.

We call this consensus protocol the “adaptive edge consensus protocol”, because it is designed for the edge network and can adapt to the workload change on the edge. The protocol divides the sequence of Paxos instances into epochs, i.e., sub-sequences of Paxos instances with a fixed length N_{epoch}. At the beginning of each epoch, the protocol examines the past running history and decides how to distribute the leadership among the edge nodes for this epoch. When an edge node receives an event from the client, it tries to commit the event in its next Paxos instance in the sequence. If an edge node has already committed an event e in its Paxos instance ins, but some Paxos instances belonging to the others and ahead of ins in the sequence have no event committed yet, the order of e is still undetermined, and ins is blocked by the Paxos instances. If ins has been blocked by the Paxos instances for a long enough time T_{skip}, the edge node will try to skip those Paxos instances. If an edge node intends to commit an event, but fails for N_{fail} times because its Paxos instances have been skipped by the others, it sends the event to the cloud. The cloud collects all such events and orders them by their arriving time. By the end of each epoch, the cloud sends the sequence of all such events it has collected in this epoch to the edge nodes. When receiving this event sequence, the edge nodes append it to their local event sequence, and close the current epoch. This ending interaction between the cloud and the edge nodes is similar but different to the other Paxos instances in the sequence, and is hence called a “quasi-Paxos instance” in our protocol.

Several details of the protocol should be highlighted. First, the leadership distribution scheme is deterministic, so at the beginning of each epoch, the edge nodes will generate the same leadership distribution without communicating with each other. This reduces the communication cost and improves the protocol performance. Second, unlike the Paxos instances, the quasi-Paxos instances cannot be skipped. This essentially sets an upper bound on the latency for the events under contentions, which is comparable to that of directly using the cloud for event ordering. Third, the edge nodes cannot start the next epoch until the quasi-Paxos instance of the current epoch is closed, meaning that all the edge nodes have to wait for the cloud at the end of each epoch. This is for the failure recovery purpose, which will be discussed later. By carefully choosing N_{epoch}, the quasi-Paxos instances will not block the protocol at all, or at least will not block the protocol for a significantly long time. Last, as the cloud is used to resolve the contentions among the edge nodes, it is generally called the arbitrator of the protocol.

```
Edge Node $i$, on receiving an event $e$
01: $committed$ := $False$
02: $cnt\_failures$ := $0$
03: while $cnt\_failures < N_{fail}$ do
04:     $ins_i$ := the sequence number of the next available Paxos instance assigned to $i$
05:     send ACCEPT messages ($ins_i$, $e$) to the other edge nodes
06:     if receive ACCEPTED messages ($ins_i$, $e$) from a majority of the edge nodes
        (including itself) then
07:         commit $e$ to $ins_i$ locally
08:         send COMMIT messages ($ins_i$, $e$) to the other edge nodes
09:         $committed$ := $True$
10:         break
11:     else
12:         skip $ins_i$ (i.e., commit $no\_ev$ to $ins_i$) locally
13:         $cnt\_failures$ := $cnt\_failures + 1$
14: if $committed$ = $False$ then
15:     send a DELEGATE message ($e$) to the cloud

Edge Node $j$, on receiving an ACCEPT message ($ins_i$, $e$)
16: if have never accepted a SKIP message ($ins_i$) then
17:     send an ACCEPTED message ($ins_i$, $e$) to $i$
18: else
19:     send a SKIPPED message ($ins_i$) to $i$

Edge Node $j$, on receiving a COMMIT message ($ins_i$, $e$)
20: commit $e$ to $ins_i$ locally
```

Figure 4.5: The algorithms involved in proposing and committing an event.

```
Edge Node $i$, when considering that its own Paxos instance $ins_i$ has blocked the
progress of consensus for a long enough time
01: skip $ins_i$ (i.e., commit $no\_ev$ to $ins_i$) locally
02: send COMMIT messages ($ins_i$, $no\_ev$) to the other edge nodes

Edge Node $j$, when believing that a Paxos instance $ins_i$ belonging to Edge Node $i$
($i \neq j$) has blocked the progress of consensus for a long enough time
03: send SKIP messages ($ins_i$) to the other edge nodes
04: if receive SKIPPED messages ($ins_i$) from a majority of the edge nodes (including itself)
    then
05:     skip $ins_i$ (i.e., commit $no\_ev$ to $ins_i$) locally
06:     send COMMIT messages ($ins_i$, $no\_ev$) to the other edge nodes
07: else if receive a COMMIT message ($ins_i$, $e$) then
08:     commit $e$ to $ins_i$ locally
09: else if receive ACCEPTED messages ($ins_i$, $e$) from a majority of the edge nodes then
10:     commit $e$ to $ins_i$ locally
11:     send COMMIT messages ($ins_i$, $e$) to the other edge nodes
12: else if receive an ACCEPTED message ($ins_i$, $e$) then
13:     send an ACCEPTED message ($ins_i$, $e$) to $i$

Edge Node $k$, on receiving a SKIP message ($ins_i$) from Edge Node $j$
14: if have received a COMMIT message ($ins_i$, $e$) then
15:     send a COMMIT message ($ins_i$, $e$) to $j$
16: else if have accepted an ACCEPT message ($ins_i$, $e$) then
17:     send an ACCEPTED message ($ins_i$, $e$) to $j$
18: else
19:     send a SKIPPED message ($ins_i$) to $j$
```

Figure 4.6: The algorithms involved in skipping an event.

```
Cloud, the main processing
01: $epoch$ := 0
02: send CLOSE messages ($epoch$, {}) to the edge nodes
03: while $True$ do
04:     $ev\_list$ := {}
05:     wait until having received CLOSED messages ($epoch$) from a majority of the edge
        nodes
06:     $epoch$ := $epoch + 1$
07:     send CLOSE messages ($epoch$, $ev\_list$) to the edge nodes
Cloud, on receiving a DELEGATE message ($e$)
08: append $e$ to $ev\_list$
Edge Node $i$, on receiving a CLOSE message ($epoch$, $ev\_list$)
09: send a CLOSED message ($epoch$) to Cloud
10: commit $ev\_list$ to the quasi-Paxos instance of $epoch$ locally
```

Figure 4.7: The algorithms related to the arbitration by the cloud.

The algorithms shown in Figure 4.5, 4.6 and 4.7 elaborate on how the protocol works when an event is proposed and committed, when an event is skipped and when the contentions of events are arbitrated by the cloud, respectively. All the quorums involved in these algorithms are a majority of the edge nodes, such that any two quorums intersect. Moreover, because at any time, at most a minority of the edge nodes may fail, there will always be some living edge nodes that can tell the outcomes of the actions that have ended. Therefore, the correctness of the consensus protocol is guaranteed.

When the current epoch has been closed, there are several ways to determines the leadership distribution for the upcoming epoch. Three schemes are designed for achieving this.

- Previous Epoch Only. When using this scheme, the protocol determines the leadership distribution completely based on the running history of the previous epoch that has just been closed. To be more specific, suppose there are N_{eff} effective (i.e., non-skipped) Paxos instances existing in the previous epoch, while $N_{e,i}$ of them belong to Edge Node i. Moreover, suppose the cloud commits N_{del} events at the end of the epoch, and $N_{d,i}$

of them are delegated by Edge Node i. In such a case, Edge Node i will be the leader of $(N_{e,i} + N_{d,i}) * N_{epoch}/(N_{eff} + N_{del})$ Paxos instances. These Paxos instances will be arranged from the beginning of the upcoming epoch, interleaving with those assigned to the other edge nodes as much as possible.

- Previous N Epochs. This scheme determines the leadership distribution by looking at the running history of the previous N epochs, denoted as Epoch N (the latest), Epoch $N-1$, ..., and Epoch 1 (the earliest), respectively, and each Epoch X ($X = 1, ..., N$) is assigned a weight $w_X = X/\sum_{I=1}^{N} I$. The protocol first calculates the committed event ratio for each Epoch X and each Edge Node i, i.e., $ratio_{X,i} = (N_{e,i} + N_{d,i})/(N_{eff} + N_{del})|_X$. Then in the upcoming epoch, Edge Node i will be the leader of $(\sum_{I=1}^{N} w_X * ratio_{X,i}) * N_{epoch}$ Paxos instances. Clearly, the Previous Epoch Only scheme is a special case of the Previous N Epoch scheme, i.e., $N = 1$.

- Previous N Epochs with Random Weights. This scheme is similar to the previous one, expect that the weights assigned to the epochs are calculated in a (pseudo-)randomized way. More specifically, for Epoch X, the protocol picks a number r_X from $\{1, 2, ..., f * N\}$ with the same probability, where $f = 1000$ is the broadening factor, and the weight assigned to Epoch X is $w_X = r_X/\sum_{I=1}^{N} r_I$. Note that this calculation is actually pseudo-randomized and hence deterministic, not violating the rules of distributing the leadership.

The first scheme is the simplest one and can rapidly adapt to the change on the workload. The drawback of this scheme is that if an edge node has experienced some bad conditions, such as network jitters, but later recovers, it may take a long time for the edge node to regain its leadership share in the epochs. The second scheme involves more previous epochs to deal with this situation, but it may suffer when the workload changes rapidly. The third scheme adopts a randomized approach, and is thus more resilient to the transient changes on the workload and on the network conditions than the second one, but suffers from the same problem as the second scheme.

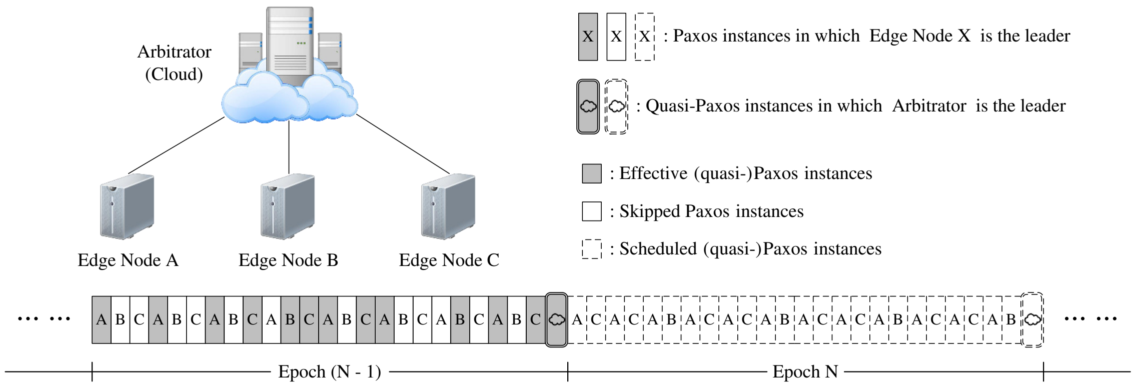

Arbitrator (Cloud)
X X X : Paxos instances in which Edge Node X is the leader
: Quasi-Paxos instances in which Arbitrator is the leader
: Effective (quasi-)Paxos instances
: Skipped Paxos instances
: Scheduled (quasi-)Paxos instances
Edge Node A
Edge Node B
Edge Node C
A B C A B C A B C A B C A B C A B C A B C A B C
A C A C A B A C A C A B A C A C A B A C A C A B
Epoch (N - 1)
Epoch N

Figure 4.8: The adaptive edge consensus protocol. The leadership distribution scheme shown in the figure is the Previous Epoch Only scheme.

Figure 4.8 illustrates how the adaptive edge consensus protocol works. Three edge nodes, i.e., Edge Node A, B and C, compose the edge network, and a cloud at the backend acts as the arbitrator. Two epochs, Epoch (N – 1) and Epoch N, are shown in the figure. Suppose the cloud does not commit any event but merely closes Epoch (N – 1) at the end of it. Because the protocol is using the Previous Epoch Only scheme to distribute the leadership, in Epoch N, Edge Node A is assigned $6 * 24/12 = 12$ Paxos instances, Edge Node B is assigned $2 * 24/12 = 4$ Paxos instances, and Edge Node C is assigned $4 * 24/12 = 8$ Paxos instances. Note that after this calculation, the Paxos instances are scattered as evenly as possible throughout Epoch N.

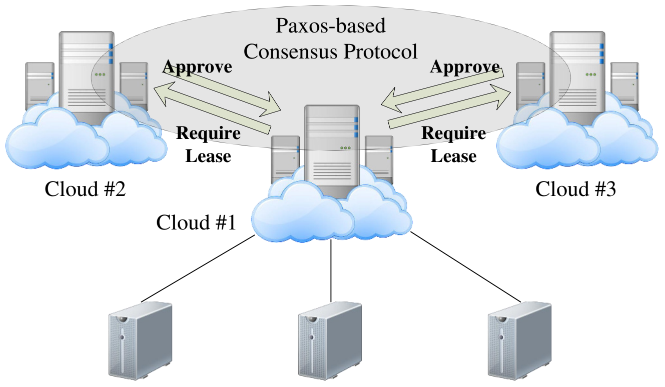
4.4.4 Working with the Clouds

As mentioned in Section 4.4.3, the adaptive edge consensus protocol is designed for systems with only one edge network and one backend cloud. When the system contains multiple edge networks and multiple backend clouds, such as those described in Section 4.3, the adaptive edge consensus protocol cannot work effectively without combining with another cloud-level protocol. For this reason, we have designed a cloud-level protocol to work around this problem.

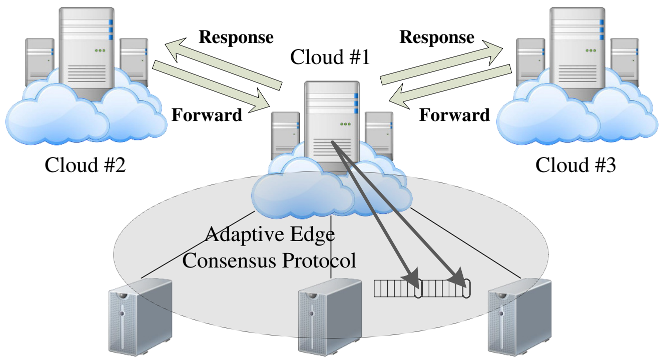
We design the cloud-level protocol as follows. The system initially works in the equality mode, i.e., all the clouds work together using a Paxos-based consensus protocol, ordering all the events across the system. Any Paxos-based consensus protocol can be used here, but the leadership-sharing ones are preferable, such as Mencius. The edge nodes are only used to forward the events they have received to their backend cloud. Because all the cloud can see all the events that have been ordered, they can learn the workload condition of the system. If a cloud notices that in the past N_{ob} events that have been ordered, more than $ratio_{thresh} * N_{ob}$ events come from the same edge network it backs, the cloud will try to make the system work in the master-slave mode. It does so by asking for a lease from the other clouds using the same consensus protocol for ordering the events. After the cloud successfully receives the lease approvals from a quorum, it informs the edge network, and the edge network will work with the cloud using the adaptive edge consensus protocol. Now the edge network and the cloud become the master, and all the other parties become the slaves, i.e., the system is working in the master-slave mode. The protocol is thus called the lease-based master-slave protocol in our solution.

Notably, when the system is working in the master-slave mode, all slaves, including both the other clouds and the other edge networks backed by the master cloud, will forward the events they have received to the master cloud. The master cloud will order all such events locally, together with those received from the master edge network for contention resolving, based on the event arriving time. It will then commit all the locally-ordered events in the upcoming quasi-Paxos instance, thus determining the global order of them. After that, the order of all the events will be broadcast by the master cloud to the slave clouds.

Figure 4.9 shows how the leased-based master-slave protocol works. It is worth mentioning that the lease can only be used by the master for N_{lease} epochs. After the N_{lease} epochs, the other parities consider that the lease is expired, and the system will return to the equality mode unless the master cloud has successfully received an extension for the lease from the other clouds. The extension will be another N_{lease} epochs, and the master can extend the lease for potentially many times. When receiving a lease request



(a) Cloud #1 observes that most workload of the system has been received by an edge network it backs. It then asks for a lease to handle the workload at that edge network, through the Paxos-based consensus protocol.



(b) The edge network receives the lease, and works with Cloud #1 using the adaptive edge consensus protocol. Cloud #2 and Cloud #3 forward the events they have received to Cloud #1, which in turn commits them into the quasi-Paxos instances.

Figure 4.9: The lease-based master-slave protocol.

or a lease-extension request, a cloud will check if it has more than N_{out} events that are not ordered yet. If the answer is yes, it will reject the request. Otherwise, it will accept the request. Clearly, the lease-based master-slave protocol can adapt to and take advantage of the highly unbalanced workload in the system, achieving low user-perceived latency in the scenarios similar to those introduced in Section 4.3. The solution proposed by this work, i.e., Nomad, is essentially the combination of the adaptive edge consensus protocol and the lease-based master-slave protocol.

4.4.5 Dealing with the Failures

As summarized in Section 4.4.2, at any time, a minority of the clouds may fail or become network-partitioned simultaneously. Similarly, at any time, a minority of the edge nodes in an edge network may fail or become network-partitioned. It is essential to guarantee the correctness of Nomad, i.e., when an event has been committed, any party in the system will not observe a contradicting state of the event at any time in the future. Otherwise, inconsistency occurs in the system, violating the design rules of Nomad.

When the system is working in the equality mode, all the clouds cooperate with each other using an existing Paxos-based consensus protocol in the literature. In such a case, the correctness of Nomad can be guaranteed by the Paxos-based consensus protocol.

When the system is working in the master-slave mode, however, two failure cases need to be taken into consideration. First, when some master edge nodes fail or become network-partitioned, the correctness of Nomad can be guaranteed. As mentioned in Section 4.4.3, all actions, including proposing an event, skipping a Paxos instance and closing an epoch, require the initial party to collect the acceptance from a majority of the master edge nodes before succeeding. Therefore, at any time, at least one living master edge node can tell the outcomes of the actions that have ended, so no contradicting state will occur. Second, when the master cloud has failed, the correctness of Nomad is still guaranteed, because the master cloud can only commit events with the acceptance of a majority of the master edge nodes.

Notably, a severe consequence resulted from the failure of the master cloud is that the process of the Nomad protocol will be completely blocked, because the master edge nodes have to synchronize with the master cloud at the end of each epoch. In contrast, other kinds of failures will not block the process of Nomad. To work around this problem, Nomad treats the slave clouds as the backups of the master cloud. When a slave node suspects that the master cloud has failed, it will directly contact the master edge nodes, asking them to accept it as the new arbitrator. After the slave node has collected the acceptance from a majority of the master edge nodes, it becomes the new arbitrator, and works with the master edge nodes until the lease expires. The correctness of this process is guaranteed by requiring the acceptance from a majority of the master edge nodes.

It should be mentioned that the aforementioned method for arbitrator failover is possible because at least one living cloud can access all the living master edge nodes at any time, as assumed in Section 4.4.2. However, a network partition in the real world may make all the master edge nodes unreachable to the outside, and if no ad-hoc solution is employed for handling this problem, such as setting a satellite network for backup, arbitrator failover cannot be accomplished. In such a case, the protocol cannot make any progress until the arbitrator recovers. Nevertheless, the correctness of consensus will not be violated, and the protocol will continue working as soon as the arbitrator has recovered.

4.5 Evaluation

To evaluate Nomad, we have implemented a prototype, and deployed it on a testbed. Experiments on the prototype shows the performance of Nomad under different situations.

4.5.1 Testbed Setup

We first build an edge-cloud testbed. Three PC servers are used as three clouds, and several laptops are used as the edge nodes that form an edge network. The edge network is backed by one of the clouds. The RTT between the edge nodes is set to 10 ms, expect a slow one, which has a 40 ms RTT to the others. The RTT between the edge nodes and their backend cloud is 60 ms. As the workload for testing is simulated, the RTT between the client and the edge is assumed to be 10 ms. The RTT between the clouds is set to 100 ms. The bandwidth between any two parties is set to 100 Mbps, and the message size is set to 1 KB across the system.

4.5.2 Performance of the Adaptive Edge Consensus Protocol

After building the testbed, we implement a prototype of the adaptive edge consensus protocol, with the three leadership distribution schemes described in Section 4.4.3, and deploy it on the testbed. N_{epoch} is set to 100. Two types of workloads are simulated. The first one is a stable one; every edge node stably receives 1,000 events per second. The second one is a changing one. Every edge node stably receives 500 events per second, and another workload, which is the sum of all these individual workloads, moves from one edge node to another in a round robin manner, with a rate of 500 events per second. For example, suppose there are five edge nodes in the system, denoted as Edge Node $1, 2, ..., 5$. In the first second, Edge Node 1 receives $500 + 500 * 5 = 3000$ events, and each of the other edge nodes receives 500 events. In the second second, Edge Node 1 receives $3000 - 500 = 2500$ events, Edge Node 2 receives $500 + 500 = 1000$ events, and each of the other edge nodes receives 500 events. In the sixth second, Edge Node 1 receives 500 events, Edge Node 2 receives 3000 events, and each of the other edge nodes receives 500 events. Then in the seventh second, the workload starts to move from Edge Node 2 to Edge Node 3, and so on. Using this workload, we simulate the typical situation of the IoT payment application.

We feed the two workload to our prototype. Five leadership distribution schemes are tested, i.e., Previous Epoch Only, Previous 5 Epochs, Previous 10 Epochs, Previous 5 Epochs with Random Weights and Previous 10 Epoch with Random Weights. For comparison purposes, we also implement Multi-Paxos, Mencius and E-Paxos, and feed the workloads to them. Note that we assume the edge network can utilize non-FIFO network links, so for Mencius, piggybacking messages is not allowed. With these settings, two groups of experiments are conducted. In the first group, the edge network contains 5 edge nodes. In the second group, it contains 7 edge nodes.

Figure 4.10 depicts the results of those experiments. Clearly, the Nomad protocol outperforms the other three protocols in all settings, especially when using the Previous

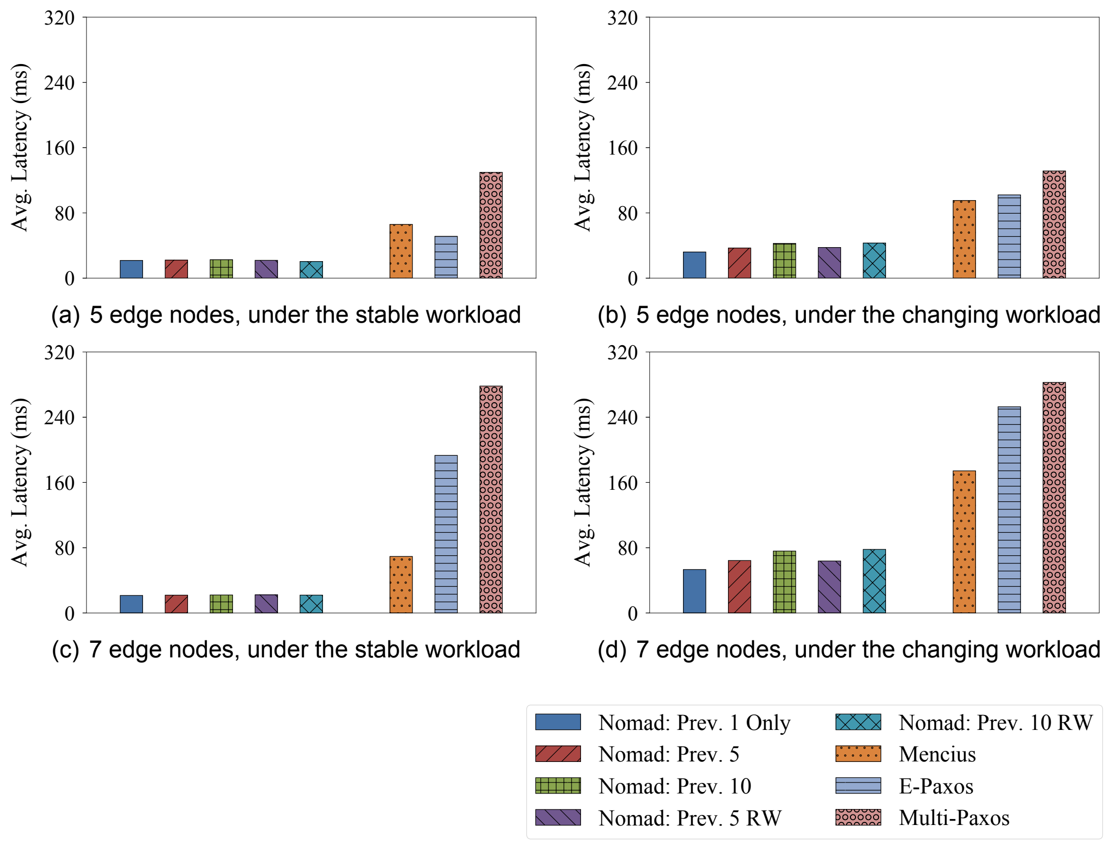


Figure 4.10: The average user-perceived latency of different consensus protocols under different settings.

Epoch Only scheme. When using other leadership distribution schemes, the average user perceived-latency is slightly larger than that of using the Previous Epoch Only scheme. As the Previous N Epoch and Previous N Epoch with Random Weights schemes are designed for fast leadership recovery, this means that they work with acceptable overhead.

4.5.3 Performance of the Leadership Distribution Schemes

To determine the effectiveness of the three leadership distribution schemes, i.e., the Previous Epoch Only scheme, the Previous N Epochs scheme and the Previous N Epochs with Random Weights scheme, we conduct the following experiment on the prototype. The edge network is configured to have five edge nodes. The workload is that in the first

50 epochs, each edge node stably receives 1,000 events per second. Then in the following 10 epochs, one edge node other than the slow one receives no event at all, while the workload on the others keeps unchanged. After that, the workload on the unloaded edge node returns to 1,000 events per second. This simulates the situation that an edge node experiences a transient problem but soon recovers. Figure 4.11 shows the changes on the leadership share of the temporally unloaded edge node caused by the changing workload.

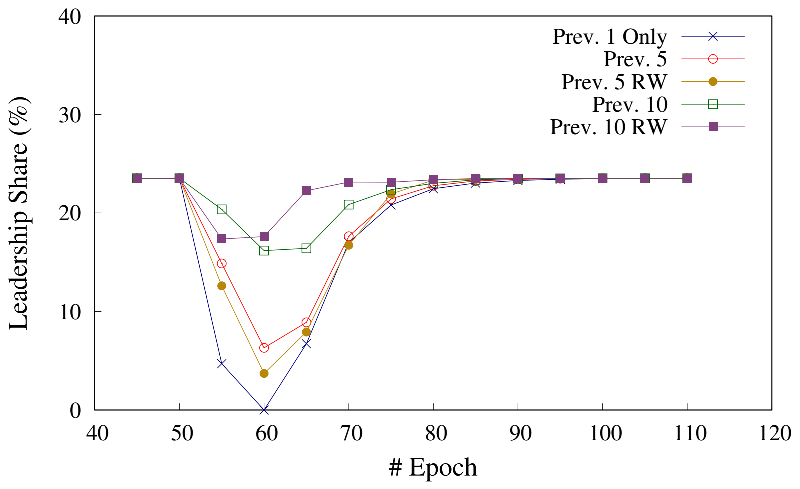


Figure 4.11: The leadership share of a temporarily unloaded edge node.

Clearly, when the Previous Epoch Only scheme is being utilized, the leadership share changes sharply with the workload change. On the other hand, when utilizing the other two schemes, the leadership share changes in a moderate manner. Moreover, the Previous 10 Epochs schemes produce the most stable outcomes, while the results of the Previous 5 Epoch schemes are in-between those of the Previous Epoch Only scheme and those of the Previous 10 Epoch schemes. This suggests that when the protocol takes more previous epochs into consideration, it can resist the transient bad conditions to a larger extent. On the other hand, this also suggests that the protocol will adapt to the workload change more rapidly when considering only one previous epoch.

4.5.4 Performance of the Leased-based Master-Slave Protocol

We also conduct an experiment to examine how the lease-based master-slave protocol works. Three PC servers are used as three clouds, denoted as Cloud #1, #2 and #3. Cloud #1 is set to be the backend cloud of the edge network. The edge network is configured to have five edge nodes. The workload is that in the first 5 seconds, every cloud receives 500 events per second. For Cloud #1, the workload is completely from the backed edge network, i.e., each edge node in the edge network receives 100 events per second. For Cloud #2 and #3, we merely simulate the condition that the events are received by a virtual edge network backed by the cloud. Then in the following 5 seconds, the five edge nodes backed by Cloud #1 receives 1,000 events per second, so Cloud #1 receives 5,000 events per second, while the workload on the other two clouds keeps unchanged. After that, the workload on Cloud #1 returns to 500 events per second. This simulates a situation similar to that of the IoT payment application. Furthermore, $ratio_{thresh}$ is set to 0.5, N_{ob} is set to 2,000, and N_{lease} is set to 10. Mencius is implemented as the consensus protocol that connects the three clouds. Figure 4.12 shows the changes on the average user-perceived latency in this process.

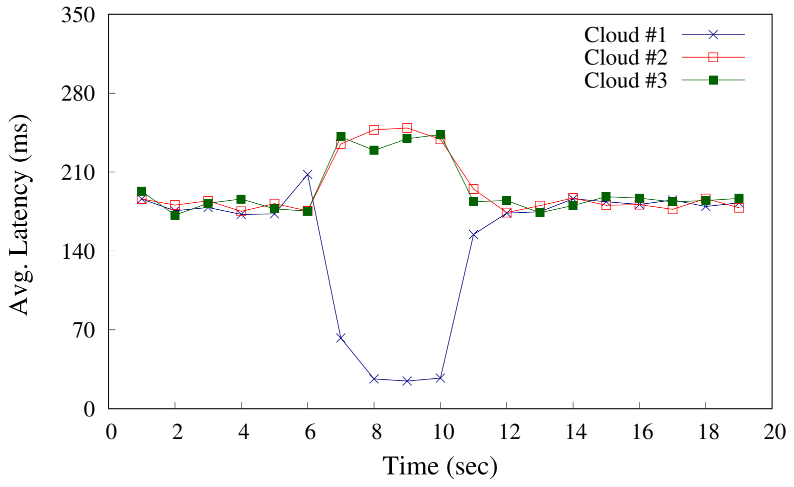


Figure 4.12: The average user-perceived latency of the three clouds.

From Figure 4.12, it can be figured out that the lease-based master-slave protocol can adapt to the change on the cloud-level workload very quickly, in about only one second for both entering and exiting the high-load phase. This proves that the lease-based master-slave protocol can work quite efficiently. Note that the lease granted to Cloud #1 is extended for several times during the high-load phase before being revoked by the system. Choosing a smaller $ratio_{thresh}$ and a smaller N_{ob} can help the protocol adapt to the workload change more quickly, but may introduce undesirable overhead if transient changes on the cloud-level workload frequently occur in the system.

4.6 Conclusions

In this work, we present Nomad, a consensus protocol achieving fast event ordering for large-scale edge-cloud applications. Nomad consists of an edge-level adaptive consensus protocol and a cloud-level master-slave protocol, which can work together to efficiently order the events received across the system. We have implemented a prototype of Nomad and deployed it on a real-world testbed. Evaluation on the prototype reveals that Nomad outperforms the existing consensus solutions in edge computing environments.

Chapter 5

Conclusions and Future Work

This dissertation presents our study on edge computing. Specifically, we aim to provide system support for end users, improving the efficiency and security of their applications.

We investigate three ways of building system support for the application scenarios in edge computing. We study: 1) how to support the existing, non-edge applications in edge computing environments without modification, 2) how to facilitate the development of new edge applications with minimal programming effort while ensuring good efficiency and robustness of the applications, and 3) how to build an efficient consensus protocol for edge computing, as consensus is the most important fundamental problem in distributed computing. We believe that building system support for edge computing in these three ways will benefit a vast majority of, if not all, application scenarios in edge computing, and the work proposed in this dissertation makes significant contributions to the state of the art of edge computing research.

Nevertheless, although less significant in our opinion, there are still some other ways of building system support for edge computing that are worth being explored. For example, how to achieve fast distributed transactions is also an important fundamental problem in distributed computing. Although it is to some degree related to the consensus problem, building an efficient transnational system for edge computing is still a challenging and interesting research mission. Additionally, although we have investigated the aforementioned three ways of building system support for edge computing, there is still plenty of work to do in fully exploring them. For example, in the first project, we only provide system support for mobile applications, because we believe that doing so will benefit a great number of users. However, other existing applications, such as those designed for cloud computing, are also worth being supported in edge computing environments. Moreover, in the second project, we have only considered several fundamental services, including task scheduling and migration, data dispatching and synchronization, and lock management. Other fundamental services, such as access control, privacy preserving, metadata management, etc., are also important in supporting some kinds of edge applications and worth being investigated.

We summarize as below the lessons we have learned from the three research projects proposed in this dissertation, which we believe will be helpful for other researchers in investigating how to build system support for edge computing.

- Identifying the characteristics distinguishing edge computing from other distributed computing paradigms is essential in designing efficient solutions for edge computing. As mentioned above, we aim to design an efficient consensus protocol for edge computing in the third project. There are many existing consensus protocols in the literature, some of which are well-designed and can achieve good performance for distributed computing in general, including edge computing. However, we always believe that an efficient, edge-computing-oriented consensus protocol will outperform the existing consensus protocols, because such a protocol can make a good fit for the unique characteristics of edge computing while the existing ones cannot. In the end, we have figured out that the network is heterogeneous in an edge-cloud computing system, which is unique to the edge computing paradigm, and we have designed a consensus protocol based on this insight. Evaluation on the consensus protocol shows that it outperforms the existing consensus protocol in edge-cloud computing environments, justifying our aforementioned statement.

- There are many available resources that can be utilized to facilitate the research on edge computing. As mentioned above, in the second project, we design a middleware running atop the edge computing infrastructure. When we evaluate our design, we need to implement a prototype of the middleware and conduct experiments on it. If we had implemented the prototype from scratch, it would take a significant amount of time and effort. Because what we really need is just to verify the effectiveness of our design, so it is important to build such a prototype with as little time and effort as possible. There are many available projects for distributed computing in general and for specialized distributed computing paradigms other than edge computing, so it is very likely that the researchers can find some candidate projects for their research, and they can simply implement their prototype systems by making minor modifications on those projects. In our case, we have implemented a prototype of the middleware based on OpenStack [75], a well-known cloud computing platform, with reasonable time and implementation effort.

To conclude this dissertation, we have made significant contributions in building system support for the edge computing paradigm. We have identified and investigated three important ways of building such system support, which will benefit a vast majority of, if not all, application scenarios in edge computing. We have also figured out the possible future work on this research topic and shared our experience on how to conduct research along this line. We believe that edge computing will become increasingly important in the future, and building system support for edge computing will bring convenience to many people and benefit the whole society.

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud*, FiCloud '14, 2014.

[2] Mohammad Aazam and Eui-Nam Huh. E-hamc: Leveraging fog computing for emergency alert service. In *Proceedings of the 2015 IEEE International Conference on Pervasive Computing and Communication Workshops*, PerCom '15 Workshops, 2015.

[3] Advanced Micro Devices. AMD Virtualization. `http://www.amd.com/en-us/solutions/servers/virtualization`, 2015.

[4] Amazon. Amazon web services (aws) - cloud computing services. `https://aws.amazon.com/`, 2019.

[5] Amazon. Elastic compute cloud. `https://aws.amazon.com/ec2/`, 2019.

[6] Android-x86 Project. `http://www.android-x86.org/`, 2015.

[7] Apache. Apache Spark. `http://spark.apache.org/`, 2016.

[8] Apache. Apache Storm. `http://storm.apache.org/`, 2016.

[9] Apache. Welcome to Apache Hadoop! `http://hadoop.apache.org/`, 2016.

[10] Ricardo A. Baratto, Shaya Potter, Gong Su, and Jason Nieh. Mobidesk: Mobile virtual desktop computing. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, MobiCom '04, 2004.

[11] Laura Belli, Simone Cirani, Gianluigi Ferrari, Lorenzo Melegari, and Marco Picone. A graph-based cloud architecture for big stream real-time applications in the internet of things. In *Proceedings of the Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2014*, ESOCC '14 Workshops, 2015.

[12] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, 2012.

[13] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '12, 2012.

[15] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[16] Yu Cao, Songqing Chen, Peng Hou, and Donald Brown. Fast: A fog computing assisted distributed analytics system to monitor fall for stroke mitigation. In *Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage*, NAS '15, 2015.

[17] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, 1999.

[18] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, 2007.

[19] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. In *Proceedings of the 10th Panhellenic Conference on Advances in Informatics*, PCI '05, 2005.

[20] Check Point. 77% of Android Devices Still Endanger Users Due to Design Flaws. `https://blog.checkpoint.com/2017/11/30/77-android-devices-still-endanger-users-due-design-flaws/`, 2017.

[21] Yuang Chen and Thomas Kunz. Performance evaluation of iot protocols under a constrained wireless access network. In *Proceedings of the 2016 International Conference on Selected Topics in Mobile Wireless Networking*, MoWNeT '16, 2016.

[22] Bin Cheng, Apostolos Papageorgiou, Flavio Cirillo, and Ernoe Kovacs. Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology. In *Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things*, WF-IoT '15, 2015.

[23] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 2011.

[24] Byung-Gon Chun and Petros Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS '09, 2009.

[25] Cisco. Fog computing and the internet of things: Extend the cloud to where the things are. White paper, Cisco, 2015.

[26] Luca Costantino, Novella Buonaccorsi, Claudio Cicconetti, and Raffaella Mambrini. Performance analysis of an lte gateway for the iot. In *Proceedings of the 2012 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WoWMoM '12, 2012.

[27] 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*, MobiSys '10, 2010.

[28] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[29] Yong Cui, Hongyi Wang, Xiuzhen Cheng, and Biao Chen. Wireless data center networking. *IEEE Wireless Communications*, 18(6):46–53, 2011.

[30] Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] Harishchandra Dubey, Jing Yang, Nick Constant, Amir Mohammad Amiri, Qing Yang, and Kunal Makodiya. Fog data: Enhancing telehealth big data through fog computing. In *Proceedings of the 4th ASE International Conference on Big Data Science and Computing*, ASE BD&SI '15, 2015.

[33] ETSI. Mobile edge computing – introductory technical white paper. White paper, ETSI, 2014.

[34] I. Farris, L. Militano, M. Nitti, L. Atzori, and A. Iera. Federated edge-assisted mobile clouds for service provisioning in heterogeneous iot environments. In *Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things*, WF-IoT '15, 2015.

[35] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[36] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Distributed data flow: A programming model for the crowdsourced internet of things. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, 2015.

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[38] Google. Google cloud. `https://cloud.google.com/`, 2019.

[39] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, 2012.

[40] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, 2014.

[41] Hao Han, Yunxin Liu, Guobin Shen, Yongguang Zhang, and Qun Li. Dozyap: Power-efficient wi-fi tethering. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, 2012.

[42] Zijiang Hao and Qun Li. Poster abstract: Edgestore: Integrating edge computing into cloud-based storage systems. In *Proceedings of the First IEEE/ACM Symposium on Edge Computing*, SEC '16, 2016.

[43] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, 2013.

[44] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology*, 65(6):3860–3873, 2016.

[45] Chin-Lung Hsu and Judy Chuan-Chuan Lin. An empirical examination of consumer adoption of internet of things services. *Computers in Human Behavior*, 62(C):516–527, 2016.

[46] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing: A key technology towards 5g. `https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf`, 2015.

[47] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '10, 2010.

[48] Krittin Intharawijitr, Katsuyoshi Iida, and Hiroyuki Koga. Analysis of fog model considering computing and communication latency in 5g cellular networks. In *Proceedings of the 2016 IEEE International Conference on Pervasive Computing and Communication Workshops*, PerCom '16 Workshops, 2016.

[49] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *Proceedings of the 2015 IEEE Conference on Open Systems*, ICOS '15, 2015.

[50] Kang Kai, Wang Cong, and Luo Tao. Fog computing for vehicular ad-hoc networks: Paradigms, scenarios,and issues. *The Journal of China Universities of Posts and Telecommunications*, 23(2):56–96, 2016.

[51] Oanh Tran Thi Kim, Nguyen Dang Tri, VanDung Nguyen, Nguyen H. Tran, and Choong Seon Hong. A shared parking model in vehicular network using fog and cloud environment. In *Proceedings of the 2015 17th Asia-Pacific Network Operations and Management Symposium*, APNOMS '15, 2015.

[52] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[53] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[54] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[55] Leslie Lamport. Generalized consensus and paxos. Technical report, MSR-TR-2005-33, Microsoft Research, 2005.

[56] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[57] Grace Lewis, Sebastián Echeverría, Soumya Simanta, Ben Bradshaw, and James Root. Tactical cloudlets: Moving cloud computing to the edge. In *Proceedings of the 2014 IEEE Military Communications Conference*, MILCOM '14, 2014.

[58] Youming Lin and Mario Di Francesco. Energy consumption of remote desktop access on mobile devices: An experimental study. In *Proceedings of the 2012 IEEE 1st International Conference on Cloud Networking*, CloudNet '12, 2012.

[59] Tom H. Luan, Longxiang Gao, Zhi Li, Yang Xiang, and Limin Sun. Fog computing: Focusing on mobile users at the edge. *arXiv:1502.01815*, 2015.

[60] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, 2004.

[61] H. Madsen, Bernard Burtschy, G. Albeanu, and Fl. Popentiu-Vladicescu. Reliability in the utility computing era: Towards reliable fog computing. In *Proceedings of the 2013 20th International Conference on Systems, Signals and Image Processing*, IWSSIP '13, 2013.

[62] Olli Mäkinen. Streaming at the edge: Local service concepts utilizing mobile edge computing. In *Proceedings of the 2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, NGMAST '15, 2015.

[63] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, 2008.

[64] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[65] Microsoft. Microsoft azure. `https://azure.microsoft.com/`, 2019.

[66] Microsoft Azure. Azure Gaming – Cloud Game Development. `https://azure.microsoft.com/en-us/solutions/gaming/`, 2018.

[67] Antti P. Miettinen and Jukka K. Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, 2010.

[68] Roberto Mijat and Andy Nightingale. Virtualization is coming to a platform near you. `https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`, 2015.

[69] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[70] Xudong Ni, Zhimin Yang, Xiaole Bai, Adam C. Champion, and Dong Xuan. Diffuser: Differentiated user access control on smartphones. In *Proceedings of the 2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, MASS '09, 2009.

[71] Takayuki Nishio, Ryoichi Shinkuma, Tatsuro Takahashi, and Narayan B. Mandayam. Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. In *Proceedings of the First International Workshop on Mobile Cloud Computing & Networking*, MobileCloud '13, 2013.

[72] Nvidia. GeForce Now. `https://www.nvidia.com/en-us/geforce/products/geforce-now/`, 2018.

[73] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, 1988.

[74] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14, 2014.

[75] OpenStack. `https://www.openstack.org/`, 2015.

[76] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, 2013.

[77] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures – a technology review. In *Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud*, FiCloud '15, 2015.

[78] Apostolos Papageorgiou, Bin Cheng, and Ernö Kovacs. Real-time data reduction at the network edge of internet-of-things systems. In *Proceedings of the 2015 11th International Conference on Network and Service Management*, CNSM '15, 2015.

[79] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, 2008.

[80] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, 2013.

[81] K. P. Saharan and Anuj Kumar. Fog in comparison to cloud: A survey. *International Journal of Computer Applications*, 122(3):10–12, 2015.

[82] Ola Salman, Imad Elhajj, Ayman Kayssi, and Ali Chehab. Edge computing enabling the internet of things. In *Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things*, WF-IoT '15, 2015.

[83] Subhadeep Sarkar and Sudip Misra. Theoretical modelling of fog computing: A green computing paradigm to support iot applications. *IET Networks*, 5(2):23–29, 2016.

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

[85] 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.

[86] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. Cloudlets: at the leading edge of mobile-cloud convergence. In *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services*, MobiCASE '14, 2014.

[87] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[88] 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*, DEBS '16, 2016.

[89] Secure Technology Alliance. Iot and payments: Current market landscape. `https://www.securetechalliance.org/wp-content/uploads/IoT-Payments-WP-Final-Nov-2017.pdf`, 2017.

[90] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.

[91] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[92] Weisong Shi and Schahram Dustdar. The promise of edge computing. *IEEE Computer Magazine*, 49(5):78–81, 2016.

[93] Ivan Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *Proceedings of the 2014 Telecommunication Networks and Applications Conference*, ATNAC '14, 2014.

[94] Ivan Stojmenovic and Sheng Wen. The fog computing paradigm: Scenarios and security issues. In *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, FedCSIS '14, 2014.

[95] Jingtao Su, Fuhong Lin, Xianwei Zhou, and Xing Lü. Steiner tree based optimal resource caching scheme in fog computing. *China Communications*, 12(8):161–168, 2015.

[96] Jin Teng, Boying Zhang, Xinfeng Li, Xiaole Bai, and Dong Xuan. E-shadow: Lubricating social interaction using mobile phones. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, 2011.

[97] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*, INFOCOM '16, 2016.

[98] Robert Triggs. How far we've come: a look at smartphone performance over the past 7 years. `https://www.androidauthority.com/smartphone-performance-improvements-timeline-626109/`, 2015.

[99] Cheng-Lin Tsao, Sandeep Kakumanu, and Raghupathy Sivakumar. Smartvnc: An effective remote computing solution for smartphones. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, 2011.

[100] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[101] Rahul Urgaonkar, Shiqiang Wang, Ting He, Murtaza Zafer, Kevin Chan, and Kin K. Leung. Dynamic service migration and workload scheduling in edge-clouds. *Performance Evaluation*, 91:205–228, 2015.

[102] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[103] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 2011.

[104] Virtual Open Systems. `http://www.virtualopensystems.com/`, 2015.

[105] Visa. Visa brings secure payment solutions to the internet of things. `https://usa.visa.com/visa-everywhere/innovation/visa-brings-secure-payments-to-internet-of-things.html`, 2018.

[106] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, 2009.

[107] Ji Wang, Xiaomin Zhu, Weidong Bao, and Ling Liu. A utility-aware approach to redundant data upload in cooperative mobile cloud. In *Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing*, CLOUD '16, 2016.

[108] Shiqiang Wang, Rahul Urgaonkar, Kevin Chan, Ting He, Murtaza Zafer, and Kin K. Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. In *Proceedings of the 2015 IEEE International Conference on Communications*, ICC '15, 2015.

[109] Tian Wang, Zhen Peng, Sheng Wen, Yongxuan Lai, Weijia Jia, Yiqiao Cai, Hui Tian, and Chen Yonghong. Reliable wireless connections for fast-moving rail users based on a chained fog structure. *Information Sciences*, 379:160–176, 2016.

[110] Wikipedia. 4G. `http://en.wikipedia.org/wiki/4G`, 2015.

[111] Wikipedia. Chrome OS. `http://en.wikipedia.org/wiki/Chrome_OS`, 2015.

[112] Wikipedia. ARM architecture. `https://en.wikipedia.org/wiki/ARM_architecture`, 2018.

[113] Wikipedia. x86. `https://en.wikipedia.org/wiki/X86`, 2018.

[114] Wikipedia. x86-64. `https://en.wikipedia.org/wiki/X86-64`, 2018.

[115] Felix Ming Fai Wong, Carlee Joe-Wong, Sangtae Ha, Zhenming Liu, and Mung Chiang. Improving user qoe for residential broadband: Adaptive traffic management at the network edge. In *Proceedings of the 2015 IEEE 23rd International Symposium on Quality of Service*, IWQoS ’15, 2015.

[116] Fengyuan Xu, Yunxin Liu, Qun Li, and Yongguang Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI ’13, 2013.

[117] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’13, 2013.

[118] Yiming Xu, V. Mahendran, and Sridhar Radhakrishnan. Towards sdn-based fog computing: Mqtt broker virtualization for effective and reliable delivery. In *Proceedings of the 2016 8th International Conference on Communication Systems and Networks*, COMSNETS ’16, 2016.

[119] Fan Ye, Raghu Ganti, Raheleh Dimaghani, Keith Grueneberg, and Seraphin Calo. Meca: Mobile edge capture and analysis middleware for social sensing applications. In *Proceedings of the 21st International Conference on World Wide Web*, WWW ’12, 2012.

[120] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Proceedings of the Third IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb ’15, 2015.

[121] 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*, MoBiData ’15, 2015.

[122] Shanhe Yi, Zhengrui Qin, and Qun Li. Security and privacy issues of fog computing: A survey. In *Proceedings of the 10th International Conference on Wireless Algorithms, Systems, and Applications*, WASA ’15, 2015.

[123] John K. Zao, Tchin-Tze Gan, Chun-Kai You, Sergio José Rodríguez Méndez, Cheng-En Chung, Yu-Te Wang, Tim Mullen, and Tzyy-Ping Jung. Augmented brain computer interaction based on fog computing and linked data. In *2014 International Conference on Intelligent Environments*, IE ’14, 2014.

[124] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP ’15, 2015.

[125] Qingyang Zhang, Zhifeng Yu, Weisong Shi, and Hong Zhong. Demo abstract: Evaps: Edge video analysis for public safety. In *Proceedings of the First IEEE/ACM Symposium on Edge Computing*, SEC '16, 2016.

[126] Yifan Zhang, Chiu Tan, and Li Qun. Cachekeeper: A system-wide web caching service for smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, 2013.

[127] Jiang Zhu, Douglas S. Chan, Mythili Suryanarayana Prabhu, Preethi Natarajan, Hao Hu, and Flavio Bonomi. Improving web sites performance using edge servers in fog computing architecture. In *Proceedings of the 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, SOSE '13, 2013.