Computers and Mathematics with Applications 63 (2012) 573–587

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

Executing mobile applications on the cloud: Framework and issues

Shih-Hao Hung, Chi-Sheng Shih, Jeng-Peng Shieh *, Chen-Pang Lee, Yi-Hsiang Huang

Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan

ARTICLE INFO

Keywords:
Smartphone
Cloud computing
Mobile network
Virtualization
Collaborative computing
Quality-of-service

ABSTRACT

Modern mobile devices, such as smartphones and tablets, have made many pervasive computing dreams come true. Still, many mobile applications do not perform well due to the shortage of resources for computation, data storage, network bandwidth, and battery capacity. While such applications can be re-designed with client–server models to benefit from cloud services, the users are no longer in full control of the application, which has become a serious concern for data security and privacy. In addition, the collaboration between a mobile device and a cloud server poses complex performance issues associated with the exchange of application state, synchronization of data, network condition, etc. In this work, a novel mobile cloud execution framework is proposed to execute mobile applications in a cloud-based virtualized execution environment controlled by mobile applications and users, with encryption and isolation to protect against eavesdropping from cloud providers. Under this framework, several efficient schemes have been developed to deal with technical issues for migrating applications and synchronizing data between execution environments. The communication issues are also addressed in the virtualization execution environment with probabilistic communication Quality-of-Service (QoS) technique to support timely application migration.

1. Introduction

Mobile and cloud computing technologies have enabled sophisticated pervasive applications. Yet, the applications on the latest generation of mobile devices today, e.g. smartphones and tablets, are still constrained by power consumption, speed of computation, size of memory, bandwidth of wireless network, etc. [1]. Since the Internet became popular, a mobile device might overcome the constraints by offloading portions of application workload onto a server machine via the network to save execution time and conserve energy [2]. Recently, cloud computing has changed software infrastructures and business models of Internet services with technologies to provide and manage abundant resources of computation and data storage over the network at relatively low amortized operation costs [3].

Today, the popularity of smart devices and mobile networks has substantially changed the way people access computers and network services. While the combination of cloud computing and mobile computing, termed *mobile cloud computing* has started to show its effects with many seemingly innovative smartphone applications and cloud services surfacing in the market today, we believe that the true potential of mobile cloud computing is yet to be explored. The following outlines the issues in today's mobile cloud computing environment from the viewpoints of the users and application developers:

- *Application re-design and deployment:* Traditional *client–server* models have been successfully used for mobile applications to leverage the resources in the cloud, but additional efforts, such as application partitioning and deployment of services, are required to enable *dynamic*, *fine-grain* client–server collaboration, where the client may decide to offload

* Corresponding author.
E-mail address: jpshieh@gmail.com (J.-P. Shieh).

0898-1221/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2011.10.044

a piece of work to the server during the runtime when the resources are available, if the offloading is beneficial. While partitioning an application to accelerate its execution by a remote server is a challenging task even for experienced programmers [4], many *web applications* or *cloud applications* actually perform slower than their local-execution version. Moreover, the client–server model is often *overkill* for deploying a personal application, as it requires someone to maintain the server and someone to pay for the service, which can incur significant cost and complexity.

- *Network condition and service availability:* The quality of the *mobile networks* is not adequate for delivering satisfactory user experiences via the collaboration between mobile devices and cloud services. It is difficult to guarantee the smoothness of a mobile application as the response time depends on the condition (i.e. latency and bandwidth) of the mobile network. For a pervasive application over a wireless network, performance and functional issues arise when the client and the server communicate over a slow or unreliable network connection. Many smartphone users share this experience and would have liked a mobile application to run locally when the network condition is poor. Unfortunately, application developers need to figure out how to achieve that by themselves, because the mechanism for making that decision is not included in any application development framework of today's smartphones. The decision process can be quite complicated, as it has to monitor the network condition, acquire the application profile, and gather information from the device and the server [1].
- *Control of applications:* With a service provider performing the computation, the users are no longer in full control of the applications. The developers and users can be trapped by proprietary interfaces and could be treated unfairly by a provider. Unlike the traditional way of making a service available by integrating applications, middleware, operating system, and server machines into a system, high-level cloud application services are available in the form of *Software as a Service* (SaaS) and *Platform as a Service* (PaaS), but compatibility has been a well-known issue with these services. After building and deploying a cloud application using one of these services, it is relatively difficult to migrate the application to another service provider. The service provider may raise charges over time or go bankrupt suddenly—a serious concern for the application developer.
- *Privacy of data:* While the online cloud service has become increasingly popular through the years, a user's privacy can be easily violated, as it is quite common for a cloud application provider to utilize user data for all sorts of claimed purposes (statistics, for example), and it is nearly impossible for users to monitor the usage of their data. The users have to trust the service providers with their personal data. This is perhaps the biggest concern to the user community, which has prevented many individual users and corporations from adopting cloud-based solutions so far. Once the data have been sent to the cloud application, it is very difficult to trace the usage of the data by the provider, especially with layers of proprietary software. Some corporations chose to build their own *private* cloud, but a private cloud has a higher total cost of ownership (TCO) than a *public cloud* and is often an overkill for small business.
- *Information security:* Offloading workload via the network increases security risk as the data propagated over the Internet and stored in the service provider could be eavesdropped by attackers in the middle and/or the service provider. While the network traffic can be protected cryptographically with secure network protocols such as secure socket layer (SSL) or virtual private network (VPN), the network structure and server organization within a public cloud expose opportunities to be explored by hackers. For example, applications belong to different owners can run on the same server, or on the two servers which can talk to each other with no firewall in between. With an Infrastructure as a Service (IaaS) provider, one may rent a virtual machine without knowing the physical machine it runs on, or other virtual machines on the same physical machine. The service provider may monitor application activities and data transfers, which is also a potential security risk. On the other side, it is a heavy burden for a service provider to ensure that user data are absolutely secure, when the user data are frequently replicated and populated all over the cloud infrastructure.

To address the above issues, we proposed a framework for a user to create a virtualized execution environment (*virtual environment* in short) in the cloud for running mobile applications. Unlike a client–server model, application redesign is not needed—the user can have an existing application running on a physical device or on a virtual environment. Our approach allows the user to control the deployment and execution of applications, and what the user needs is a trustworthy service provider to host the virtual environment, which is far more practical than verifying a growing number of service providers. In addition to offloading workload from a physical environment, the virtual environment presents opportunities to enhance the functionalities of the execution environment. For example, file sharing, automatic data backup and virus checking are additional functions that could be performed by the virtual environment in the background.

For mobile applications, the communication cost for migrating a process [5] or a virtual machine [6] can be prohibitively high. To accelerate live migration for mobile applications, we used Android [7] as our case study and developed several strategies. Being able to support live migration of existing Android applications without code modifications was a challenge for us. First, an innovative coarse-grain application migration mechanism was developed based on the application-level state-saving mechanisms supported by the Android operating environment. Second, we further categorized the types of application data to decide on the necessity and the priority for data synchronization.

For mobile devices and cloud servers to work collaboratively without jitter, being aware of the quality of communication channels and providing the quality-of-service (QoS) guarantee is one of the fundamental requirements. Toward collaborative computing [8], we believe that the execution framework can be enhanced by a QoS guaranteed communication framework in our approach. To provide better compatibility to the operating systems and application, the communication framework creates several virtual network devices in the virtualized framework. Each of the virtual network devices provides a specific

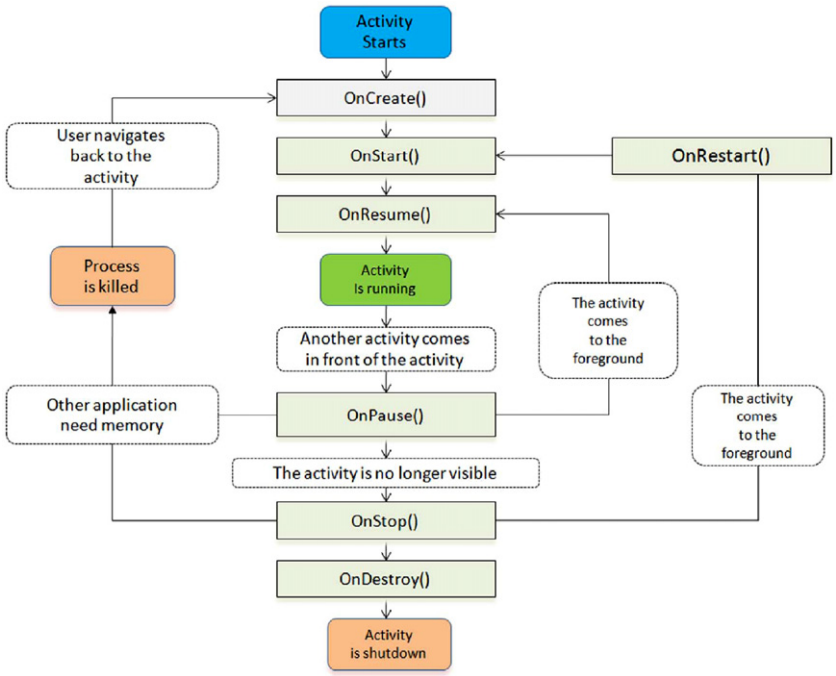

Fig. 1. Illustration of the lifecycle for an Android application activity.

QoS guaranteed communication channel. Hence, the control messages can be transmitted over a low-bandwidth channel with high transmission probability.

The rest of the paper further elaborates our approach and discusses the issues we observed in our work. Section 2 covers the background of our work by introducing the Android development framework and discussing the related research works. Section 3 discusses the framework and the virtual environment that we proposed to enhance Android applications. The system architectures and design of the probabilistic QoS guaranteed communication framework are covered in Section 4. Finally, we summarize our findings and conclude this paper in Section 5.

2. Background and related works

2.1. The Android development framework

By July 2010, there were more than 100,000 applications developed for Android as claimed by AndroLib [9]. As an optimization for smartphone applications, the Android operating environment manages applications differently from traditional operating environments. An important aspect is that the Android system may kill a process when the system falls short of memory. To decide which processes to kill, Android places each process into an importance hierarchy with the running components and its states. The five levels in the hierarchy are listed in the following, with their importance from high to low: (1) A foreground process interacts with the user, and they would be killed only when the available memory on the system is too low. (2) A visible process does not have any components, but affects what the user sees. It is still important to keep it alive unless necessary to free resource for foreground one. (3) A service process is a running service that does not belong to the above two categories. They will be killed to favor two higher processes. (4) A background process holds an activity invisible to the user, which does not have any direct impact on the user experience. It would be killed to reclaim resource for the above three types of processes with an LRU list to ensure that the most recent activities would not be killed. (5) An empty process does not associate with active components, which are often killed to balance the system workload.

The life cycle of an activity is illustrated in Fig. 1, where seven methods are involved with three nested loops:

- The entire lifetime of an application begins with the *onCreate()* method to perform initialization and ends with the *onDestroy()* method to release allocated resources.
- The visible lifetime is between *onStart()* and *OnStop()*, where the user can see the activity on-screen, whether the application is in the foreground or not.
- The foreground lifetime is between *onResume()* and *onPause()*, which is the point we are interested at. During this time, the activity is active and shown on the screen and is interacting with the user.

To conserve the energy on a smartphone, the Android operating environment suspends an application when the smartphone goes to sleep or when a new activity is issued. When an application receives the request to suspend, its *onPause()* method is called to commit unsaved state changes to persistent data and stop animations and other operations that may consume CPU time. When an activity gets the focus by user's action or a new intent is delivered, its *onResume()* method is called. Since the application may be killed by the Android operating environment during the suspension time, the *onResume()* method should include the instructions to restore the application state before the activity gets ready to receive input from the user. Application developers are advised to use the pause-resume scheme provided by the Android to save application states in the persistent storage so that the application can resume later. Since most Android applications follow this programming paradigm, we leveraged the application pause-resume scheme to design our application migration scheme.

2.2. Related works

It was shown in earlier works that *remote execution* had the potential to save the power consumption and accelerate the speed for applications running on weak devices [1,2]. *Partitioning* applications to take advantage of remote execution was actively researched. Spectra [10] proposed to monitor the current resource availability and dynamically determine the best remote execution plan for an application. Cyber foraging [11] used surrogates to improve the performance of interactive applications and distributed file systems on mobile clients. MAUI [12] proposed to reduce the programming efforts by automating program partitioning with the combination of code portability, serialization, reflection, and type safety.

Without application partitioning, *process migration* and *virtual machine migration* are two common methods for migrating the execution of a live application across the network. The ISR system [6] emulated the capabilities of suspend/resume functions in a computer system and migrated the system by storing the snapshot image of a virtual machine in a distributed storage system. Zap [5] introduced a pod (PrOcess Domain) abstraction, which provided a collection of processes with a host-independent virtualized view of the operating system, so as to support a general-purpose process migration functionality. Live migration [13] achieved rapid movement of workloads within clusters and data centers with minimal service downtimes by continuously transmitting the changes in a virtual machine to another system, but at the cost of communication overhead, which would be prohibitively high for mobile networks.

Replay systems have been researched to reduce the communication overhead in migrating virtual machines and are recently considered for mobile cloud computing [14]. ReVirt [15] replayed a long-term execution of the virtual machine instruction-by-instruction after capturing a complete log, but the scheme generated large log files and required extensive modifications to the virtual machine monitor. ReTrace [16], a trace collection tool, reduced the run-time overhead and the size of the log file by *deterministic replay* technology and file compression. Surie et al. proposed a scheme called *opportunistic replay* [17] to support virtual machine migration over a low-bandwidth network by capturing user interactions at the graphics user interface (GUI) level to reduce the replay log and ship the dirty disk chunks in reverse order to eliminate the possibility of state divergence. Flinn and Mao [14] discussed the use of deterministic replay to support mobile computing and concluded that the technology can be beneficial when applied to mobile phones. In line with Flinn's view, we also started to integrate deterministic replay into our framework.

However, few of the aforementioned previous works addressed the need for smartphone applications to offload workloads in a pervasive environment with limited network bandwidth. Nor did they address the control/privacy/secure issues surfaced in today's cloud services as extensively we did in our framework. While virtual machines [18] have been widely used to isolate individual applications consolidate servers, and amortize the operation costs, a variety of secure threats have been raised by virtual computing environments [19]. Meanwhile, virtualization could support the detection of vulnerabilities [20] and network intrusion [21].

There have been various efforts to support network QoS guarantee including IntServ/DiffServ, RSVP [22], and those adapting the IntServ and DiffServ architectures [23] and RSVP [24] for QoS in communication networks. Resource Reservation Protocol (RSVP) is an IETF-defined (RFC 2205) signaling protocol that uses Integrated Services (IntServ) to convey QoS requests to the network. The IntServ architecture specifies extensions to the best-effort traffic model—the standard delivery model used in most IP networks and the Internet. IntServ provides for special handling of priority-marked traffic, and a mechanism by which QoS-aware applications can choose service levels for traffic delivery: controlled load or guaranteed service.

RSVP is well suited to both mission-critical applications and session-oriented applications. Both applications exchange QoS data between fixed end nodes for some degree of persistence. These types of applications tend to stream data. RSVP is primarily for use with IP traffic, operating on top of IPv4 or IPv6, whereas a transport protocol resides in the protocol stack. It configures reservations for data flows along a data path predetermined by the network routing protocol.

The aforementioned works reserve network bandwidth on the routing path to support QoS for network communication. However, they do not guarantee that the operating system can send out or receive the packets on time to take advantage of the reserved bandwidth. de Niz and Rajkumar [25] developed a mechanism in operating systems to support real-time communication for Java applications. This mechanism took advantage of the resource reservation model to support network bandwidth reservation in virtual machine. However, this mechanism requires a special designed programming model and is not generally applicable. In the work, we attempt to design the framework that provides QoS for communication on system level. Hence, there is no limitation on the applicable programming model and the operating system can control the resource

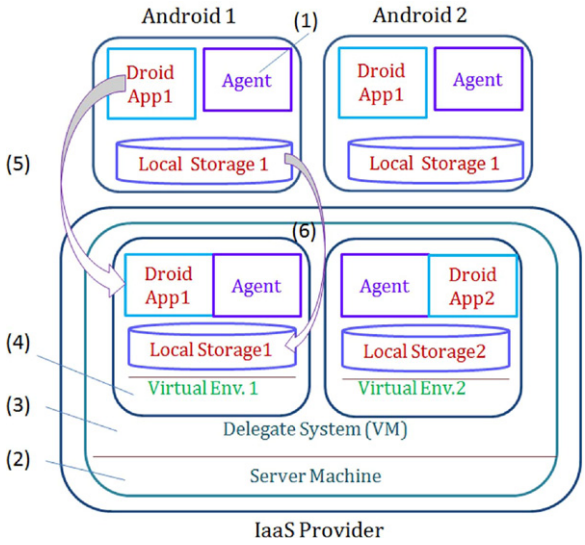

Fig. 2. Creating a virtual environment.

usage in the system. In a virtual network devices model, we will provide a programmer friendly interface to bridge QoS requirements on the application layer and those on the MAC layer.

3. A virtual environment for Android applications

Imagine a cloud-based virtual environment that is capable of running the same set of applications as the mobile device in a user's hand and shares data storage. The user may use either the mobile device or the virtual environment to execute an application or have the application migrate between two environments. The virtual environment helps offload intensive workload from the mobile device as it accelerates computation, data access, and network operations. The user may even migrate an application from the virtual environment to a personal computer, so that certain interactive tasks can be done more quickly on a large display. However, there are several key issues in designing a framework to facilitate the aforementioned scenario: How to clone the physical environment and create a virtual one? How to minimize the time required to migrate an application over a mobile network? How to minimize the costs for sharing and synchronization of data between two environments? How to secure the virtual environment?

In this section, we introduce the framework that we developed for Android users to offload applications to virtual environments in the cloud. The framework automates the creation of a virtual environment and migrates live Android application faster than traditional methods. Compared to a conventional scheme, our approach does not require the developers to redesign their applications, and we offer several effective techniques to migrate applications and data over a mobile network, with security measures included to address security and privacy issues.

3.1. Our proposed framework

In our migration framework, as illustrated in Fig. 2, we proposed the following procedures for creating such a virtual environment on a server machine with an infrastructure as a service (IaaS) provider:

1. *Installing our agent program:* The user simply installs and runs our agent program, which automates the rest of the procedures. The agent also provides the interface for the user and applications to interact with the virtual environment.
2. *Allocation of a delegate system:* The agent allocates a delegate system to host the virtual environment by subscribing to a virtual machine from an IaaS provider. The delegate system may host multiple virtual environments to save the operation cost.
3. *Setting up a virtual environment:* The agent sets up a virtual environment (a.k.a. virtual phone) on the delegate system to emulate an Android phone. For compatibility, the virtual phone needs to emulate the details of a physical Android device as much as possible.
4. *Cloning of the operating environment:* The agent uses a standard image stored in the delegate system to create a fresh virtual environment and copies the applications and data from the physical phone. An exact clone of the operating environment should increase the compatibility for applications that requires vendor-specific libraries or system services.
5. *Migration of applications:* The agent on the physical phone takes commands from the user and communicates with the agent on the virtual environment to control the operation of the virtual environment. The user of an application (or the application itself) may request the agent to migrate the application between two phones.

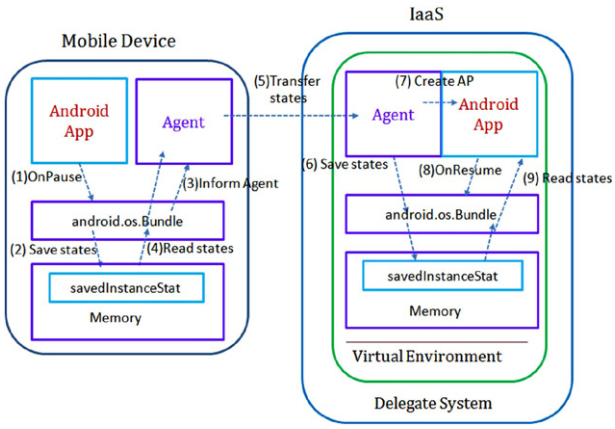

Fig. 3. Migrating an Android application.

Table 1
The sizes of the files in tested Android applications.

Applications	Package	Config.	Database	Cache buffer	State
Sudoku	740,888	146	0	0	183
AndFTP	562,795	0	0	0	1464
aBTC	92,432	0	0	0	144
YouTube	328,622	0	0	397,622	577
Music download	438,256	0	0	0	119
Music player	339,323	0	0	0	247
Google translate	951,909	0	0	0	0
Android MMS	356,392	711	0	0	0
Google gmail	489,403	0	0	0	0
Contacts provider	0	0	57,344	0	0
Average	401,273	204	5,213	36,147	249
					Unit: Byte

6. *Synchronization of applications and user data:* The agent programs on both phones collaborate to keep the application packages and user data consistent and coherent on both phones. Since continuous mirroring of files would generate a large amount of network traffics, the policies and protocols of synchronization are critical.

Note that the virtualized execution environment can be hosted by a personal computer as well. A cloud-based environment would offer better cost/performance by hosting many virtualized environments on the same server machine to amortize the costs. A personal server could be used in cases the privacy of the application is very critical. As we will discuss later in Section 3.8, the performance of a low-end personal computer can outperform the fastest smartphones by several times, finding a suitable computer to host a virtual environment would heavily depend on the applications and network conditions in practice, which was why our framework would like to address both practice issues.

3.2. Migrating an application

For migrating an application, a traditional virtual machine-based scheme needs to save and transfer the entire state of a virtual machine, which consists of the contents used by other applications in the shared memory. Depending the size of the memory, the amount of data can be hundreds of MB's in the case of an Android smartphone or several GB's for laptop computers. In comparison, our scheme only needs to transfer the state saved explicitly by the application, which costs far less than transferring the entire machine state as shown in Table 1. Basically, we pause an application on one device, send the state data files saved by the application as it enters the pause state, and resume the application on another device. As the state data files are usually small, it results in a low migration overhead.

The procedure for migrating an application is illustrated in Fig. 3. On the left-hand side: (1) The agent sends a signal to the application and has the application enter the *OnPause* function. (2) The application saves its states in the *OnPause* function and (3) informs the agent when the states are saved. (4) The agent reads the states and (5) sends the states to the agent on the other side. Then, on the right-hand side: (6) The agent saves the states and (7) starts the application (or copies the application from the other side if it does not exist). (8) The application resumes by calling the *OnResume* function and (9) resumes the execution after restoring the application state.

Obviously, this approach is applicable only when the application utilizes the Android Framework to save application state. The developers are motivated to save application state on a mobile device for two reasons: (1) For a *long-running* application, it would be risky for the application not to save its state, since the application can be terminated by an external

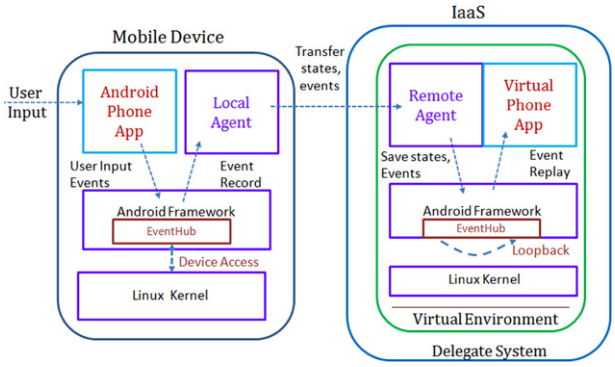

Fig. 4. Illustration of event replay architecture.

event occurring on the Android system (e.g. out of memory, out of battery, etc.). Without saving the state, it will have to restart from the beginning. (2) When an application is terminated unexpectedly, the application would lose the *input data* from the user, which could be a much worse problem than wasting time. These are inherent issues which applications on many resource-constrained mobile devices have to face, and therefore we believe that more and more mobile applications will follow Android's design guideline and perform *application-level checkpointing* during the course of execution.

3.3. Input events and application replay

As mentioned in the previous section, losing input data from the user could be much worse than wasting time. This problem can be implicitly solved by application-level checkpointing when the input data are saved as part of the application state. However, what if an input event occurs in the middle of a memory-intensive task, when the cost would be too high to save the entire application state? To deal with this dilemma, we further enhance the aforementioned state-saving scheme and integrate *application replay* techniques [14,16,17] into our framework.

Many applications are organized in *phases*, and it would be wise for such an application to save its state in between the phases when the state is less. For example, a computer game usually saves its state when the player finishes a stage. If the game crashes in the middle of a stage, the player has to start from the beginning of the stage. Another example is when an application makes a function call to a linear equation solver, the local variables and the matrices dynamically allocated in the function would significantly increase the size of the application state. When the function call ends, the size of the application state is reduced as those variables are freed. If the application iteratively calls this solver, it would be a good option to save the application state between the function calls.

In these examples, the programmer may separate the application state into two parts: A *global* state which defines the domain of the problem and control the flow of the application, and a *local* state which consists of the local data structures needed by a function. Usually, the global state is formed by global data structure, and the programmer should be able to identify these data structures. Hence, to reduce the cost of state saving, the programmer may choose to save the global state, but not the local state, in the *OnPause* function of an Android application. Thus, when the application is suspended, the application may restore its global state and resume to the last checkpoint. However, the work done by the user since the last checkpoint is lost. Therefore, it is desirable to have a record/replay mechanism which records the input events from the user in between the checkpoints and feeds the input events to the application in case the application resumes from the last checkpoint.

As shown in Fig. 4, we instrumented the *EventHub* module in the Android Framework to assist the application migration agent in monitoring and recording the input events from the user. If an input event is *non-deterministic*, then the agent records the location of the event in the program, the input data, and the time stamp in a buffer. Non-deterministic input events refer to those input events which cannot be reproduced later deterministically, e.g. keyboard input, click of a window button, voice input, camera input, etc. [16]. On the other hand, input events such as reading a file or retrieving a message from the network server can be considered as *deterministic events*, and the agent need not record these events because the same file or message can be obtained in the future.

For the replay scheme to work, the application has to *explicitly* notify the agent about a *pseudo checkpoint*, so that the agent knows when to start recording the input events. Unlike a real checkpoint, a pseudo checkpoint is simply a place holder which marks the location of resumption without actually saving the state. Instead of saving the state immediately at the pseudo checkpoint, the application defers saving of the global state until its *OnPause* function is called. If the application is paused and resumed, it will resume from the pseudo checkpoint. The purpose of pseudo checkpointing is for the agent to identify the input events needed to be replayed when the application resumes. The agent flushes the input events recorded prior to the pseudo checkpoint and start recording new events. The use of pseudo checkpointing helps reduce the overhead, since the application does not have to save its state unless it is suspended.

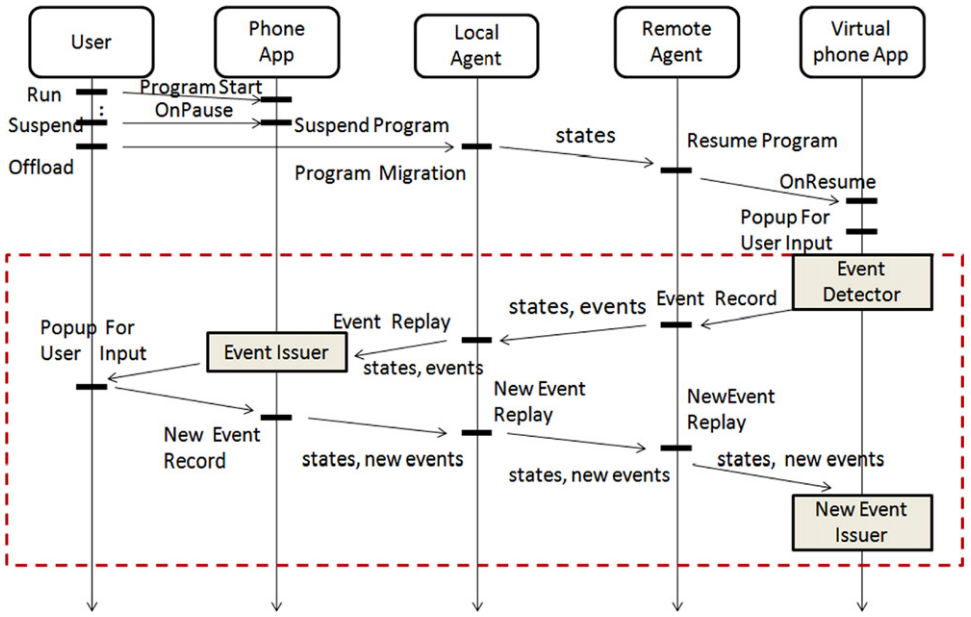

Fig. 5. A basic work flow for interactive applications.

To migrate an application via this combined scheme, the agent first suspends the application, which forces the application to perform its *OnPause* function and save its global state. The agent then transfers the state file and the recorded input events to the remote agent. The remote agent resumes the application execution and replay the recorded input events to the application to bring the application to the point of migration.

The aforementioned scheme should solve the problem of data loss without requiring the application to save its state when the cost is prohibitively high. This is also a practical solution since similar replay techniques have been used for migrating virtual machines across servers [16,17]. We proposed the pseudo checkpointing technique to reduce the state-saving overhead, but it requires the application developer to mark the the pseudo checkpoints and identify the global state in the program by inserting function calls to the emphpseudo_checkpoint() function in our library. Also, instead of migrating an application in the middle of the data-intensive function, the remote application should resume from the pseudo checkpoint to save the communication cost. We think that it is practical to extend an existing application framework such as the Android to support this pseudo checkpointing technique, as it is an effective scheme to reduce the cost of state saving, not only for migrating applications, but also for preventing data loss on a mobile device. In the future, the pseudo checkpoint technique could also be implemented into the Dalvik virtual machine as an automatic mechanism.

3.4. Interactive applications

One important difference between a physical device and a virtualized environment is the ability to receive input from the user. How does an application receive input from the user and display the results when it is executed in a remote virtualized environment? It is typical for an Android application to create an *user interface* (UI) thread to interact with the user. CloneCloud [1] solved a similar problem by partitioning the application so that the UI remains on the physical device, and the rest of the application may run remotely. In this case, the system was responsible for delivering the input events to the remote execution threads.

In our framework, we aim to address this issue without partitioning the application in advance. The framework reuses the application replay mechanism mentioned in the previous subsection to support interactive applications. The migration agent monitors the occurrence of input events and UI threads for the application executing in the virtualized environment. When a non-deterministic input event is detected, the agent first checks the event buffer and feeds the input event to the application. If the event buffer is empty for an input event, or when the application wants to display the UI window, the agent needs a mechanism to display the UI windows on the physical device. We discuss and compare three solutions here. The first solution is to migrate the application back to the physical device to obtain the input event and then migrate the application to the virtualized environment. The second solution is to send only the UI window back to the physical device to receive the input from the user via the agents. The third solution is to display the UI window via an open remote display protocol such as VNC [26].

The first solution leverages existing migration framework with event replay straightforwardly. Fig. 5 illustrates a basic work flow between the user and the four key components involved in the mechanism, i.e. phone application, local agent, remote agent, and virtual phone application. The application is started on the physical phone and migrated to the virtual

phone by the user. After a while, a window is popped up by the application to ask the user for input. The remote agent detects this input event and migrates the application back to the physical phone. After the application gets an input from the user, it is again migrated back to the virtual phone. This method works best when the application state is small and/or when non-deterministic input events occur infrequently.

The second solution requires the agent on the virtual phone to send the data structures which describes the UI window to the physical phone. In this case, the agent displays the same UI window to the user only to receive the input for the application. Once the input event is received from the user, the agent transfers the input event to the virtual phone and uses the replay scheme described in the previous subsection to feed the input event to the application. By transferring only the high-level objects and events, this solution avoids transferring the application state, which could significantly reduce the network traffic, but its implementation is highly dependent on the display protocol and cannot be easily ported to another system.

The third solution is similar to the second solution except that it uses an open remote display protocol. There is an open-source Android-based VNC server project, called *android-vnc*, which could serve the purpose. The advantage of using the popular VNC protocol is the ability to display the input window as well as the results on a variety of platforms. However, since the physical phone acts as a display terminal in this case, the responsiveness of the UI window depends heavily on the latency of the network, which can be an issue over a slow mobile network.

3.5. Native code and performance

Although Google has intended to keep Android applications platform-independent by offering a JAVA application framework, some Android applications are linked to proprietary C or assembly functions for performance reasons. These proprietary C functions become platform-specific as they are compiled into machine binaries and ship with the applications as *native libraries*. This phenomenon causes incompatibility issues for Android applications across platforms. In our study, we downloaded 2317 Android application packages and found that 5.82% of the applications were linked to proprietary native libraries. The average size for these native libraries was about 630 kB.

How would this phenomenon impact the performance of a virtualized environment? First, since *x*86-based servers are the current *de facto* standard in the cloud, we have the virtualized environment runs on *x*86-based servers with an Android environment built for *x*86, a.k.a. *Android-x86*. Since *x*86-based servers are much faster than the mobile devices built with less powerful processors, those 94.18% Android applications which are not linked to proprietary libraries should be effectively accelerated on the virtualized environment.

For the other 5.82% applications which are linked to proprietary native libraries, we may execute them via a processor emulator, such as QEMU [27], on the *x*86-based server. Or, we may find a server of the same instruction-set architecture and deploy a virtualized environment on that server. Either way, these applications would benefit less from the virtualized environment. Since the speed is very important to the application, the best solution is to have a version of the native library built for the *x*86 platform by the developer, so that the application can benefit from an *x*86-based virtual environment This somewhat complicates the distribution and management of application packages and libraries, but this may overcome the performance issue.

3.6. Synchronizing data

While an Android application is migrated from one machine to another machine, it is necessary to clone its application states and data in the storage system. A brute-force way for synchronizing data between two environments is to maintain identical data for all files in their file systems. For that, the data must be updated as soon as one environment makes changes to any file, which incurs network traffics, and the applications need to wait for the completion of any synchronization operation if a strict synchronization protocol [28] is used, which would cause a long delay with a mobile network. To mitigate this problem, we need to look further into the policies for synchronization.

First, we divide the data storage into three categories: system image, system-wide data, and application data. When a virtual environment is initialized, its filesystem is loaded with the system image and application packages that are on a standard operating environment image. Synchronizing system image and application packages happens infrequently and should not be an issue for the virtual environment in the cloud as it may download the image from a high-bandwidth network. System-wide data refers to those files that record system-wide information and/or would affect the operations of the system and applications. Libraries are examples of system data. Modifying a library may affect multiple applications running on the system. When a physical environment makes a change to its system data, it often stops the applications that might be affected and sometimes requires the system to reboot. Similarly, its virtual environment counterpart should follow the same procedure.

Application data refers to the files owned by applications, and the synchronization of application data can be done on per-application basis during the course of application migration. Thus, we apply lazy and on-demand policies [29] to synchronize the data upon the request of an application without keeping application data updated all the time with a coherence protocol. Running a collaborative application that executes simultaneously on multiple devices requires the application developer to design the data communication or synchronization scheme specifically for the application. We are working to provide an application programming interface to support collaborative applications in our framework.

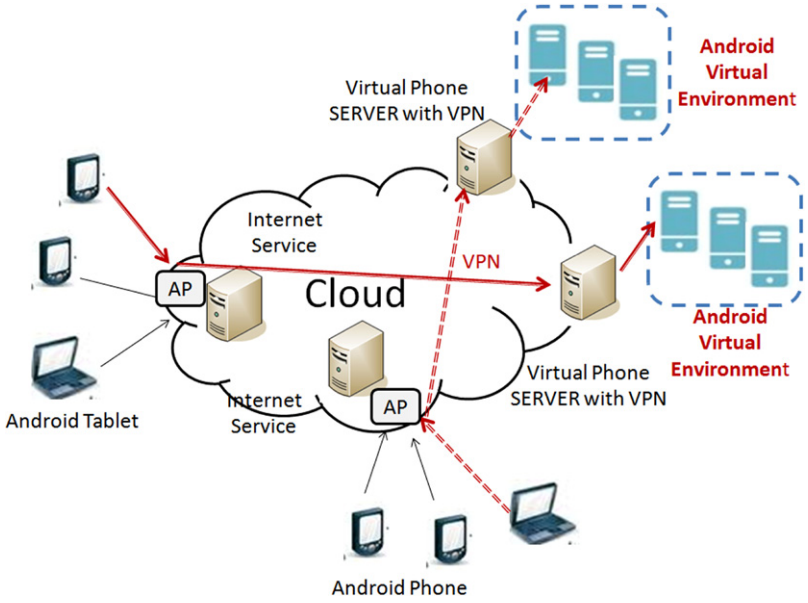

Fig. 6. A VPN connection to virtual phone.

3.7. Security and privacy measures

Since a virtual environment may operate in a public cloud, it is important to protect the user data with an end-to-end secure communication channel. In addition, choosing a trustworthy IaaS provider is critical to deploy a virtual environment. A strong authentication and encryption scheme is needed to set up the secure storage and the secure channel for storing private keys and exchanging master encryption keys. VPN is one popular connection solution for enterprise mobile users. Currently Android phones have built-in support for four different protocols to connect VPN server, namely PPTP (Point-to-Point Tunneling Protocol), L2TP (Layer 2 Tunneling Protocol), L2TP/IPSec PSK (PreSharedKey) and L2TP/IPSec CRT (Certificate) [30], depending on the security requirement. On the server side, OpenVPN [31] is an open source software package used by our framework. Fig. 6 shows the VPN scheme used in our framework, with a server working as a VPN hub to establish a private/secure tunnel initially. During the deployment stage, an end-to-end secure channel between each pair of Android phone and virtualized environment is further established. By default, we chose the L2TP/IPsec PSK protocol and distribute the pre-shared keys to the user via a separate secure channel. The scheme works with SIM cards or existing authentication devices on mobile phones. For maximum security, we may opt to distribute certificates of the servers/virtualized phones to the users via SSL.

For stronger level of trust, Trusted Platform Module (TPM) could be integrated to enhance the security on the server system, offering hardware mechanisms to store the encryption keys and perform cryptographic operations on sensitive data. To further prevent the intervention from the service provider or attackers, sensitive data in the memory and the storage could be encrypted. Since the files are encrypted and hashed, attackers from another virtual machine on the same host or in the middle of the network will find it harder to retrieve and manipulate the contents of the files. Again, TPM could be used to store the master encryption keys and perform the encryption procedure to keep the encryption keys away from the eavesdroppers.

3.8. Performance evaluation

We first used a peer-to-peer (P2P) file exchange application program, androidtorrent [32], as an example to illustrate the application migration procedure. P2P is a distributed network for participants to share their resources without a centralized coordinator. The working set in the P2P application can be quite large as the user exchanges many files with many peers, which also consumes a lot of resources on the processor, the storage, and the network. It is obvious that the workload would better be handled by a virtual environment in the cloud. The agent transferred the states saved by androidtorrent, approximately 320 kB of data, to the virtual environment in a few seconds over a 3G mobile network. In our experiment, androidtorrent was migrated to the virtual environment and resumed execution in 3.8 s, and we defined a policy for the agent not to synchronize the temporal files (i.e. incomplete downloads) as these data in these files should be re-acquired anyway.

We have further studied other Android applications to characterize the costs of migration. The files associated with an Android application can be categorized into five types: program package, configuration files, database files, cache buffers, and saved state files. Table 1 shows the sizes for these files in the applications that we examined.

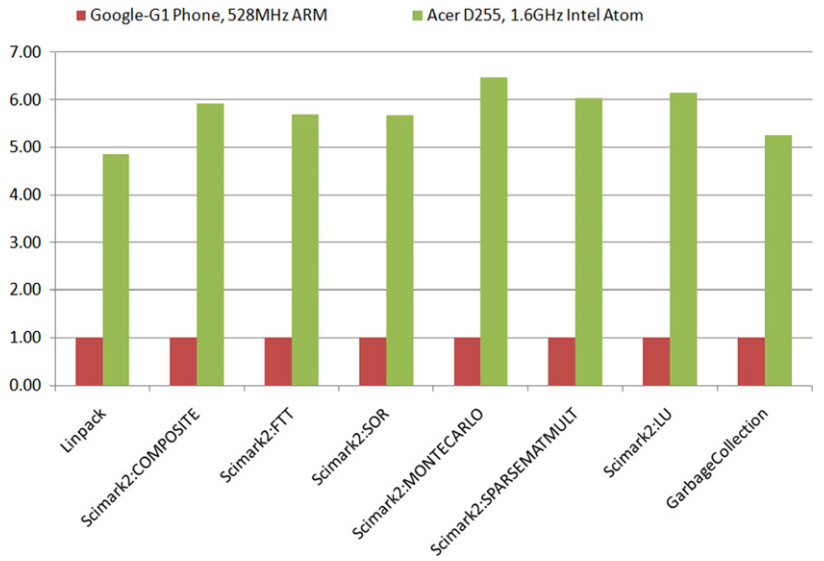

Fig. 7. Performance comparison between a physical device and a virtualized environment.

To migrate a new application to the virtual phone, the package has to be transferred only once. The configuration files and the database files are monitored closely in our framework, so any modifications to these files are synchronized immediately via the shared storage. For example, Contact Provider is part of the Android middle-ware that maintains a database for applications to find contact information, and it is important to synchronize the database files even though Contact Provider is not running on the virtual phone. The cache buffers are primarily used by streaming multimedia applications, such as YouTube, to prefetch multimedia contents, which can be discarded and reloaded by the application. Finally, the state files for an application need to be synchronized only when the application is being suspended and migrated onto another phone.

As shown in Table 1, the size of a program package varies from 92 to 951 kB, which would take a few seconds to transfer via a mobile network. As the configuration files are quite small and the contact database should not be updated frequently by a typical application, the communication costs for synchronizing files in these two categories are usually not an issue. Since the contents in cache buffers are constantly changing and can be re-fetched after being thrown away, our framework label these files as no-need-for-synchronization to reduce the communication costs. Finally, the costs for transferring state files are minimal. Web-based applications such as Google Translate, Android MMS, and Gmail are state-less without any state files at all, while the other applications have small state files that are less than 1.5 kB, which means our approach can migrate the execution of any tested application onto a virtualized environment as quickly as sending 1.5 kB over the network.

Next, we show that a virtualized execution environment accelerates the execution of mobile applications. In our experiment, we used the Intel Atom-based system host Android-*x*86 as virtual environment in the cloud. As shown in Fig. 7 the 1.6 GHz Intel Atom processor was already 4.9–6.4 times faster than the 528 MHz ARM processor on the Android phone. The results suggested that the virtual environments powered by servers equipped with low-end processors still provided sufficient performance for average applications. Migrating an application via a traditional approach, such as [6], would need to transfer the state of the entire environment by taking a snapshot of the memory. Assuming an Android environment with 512 MB of system memory, it would take more than one hour to transfer the snapshot. With our approach, it would only take milliseconds to transfer the state files saved by Android applications.

Finally, Fig. 8 shows the results of a Face Recognition application running between a HTC Desire Android phone and an Intel i7-2600 with Android-*X*86 virtual machine, migrating via the WiFi network. The program matches a human face against the faces stored in a database file and find the best matches. The curves represent three cases of execution: standalone execution on the mobile phone, standalone execution on the server, and collaborative execution. The application runs 4*x*–8*x* faster on the server, depending on the size of the face image and the size of the database file. For the collaborative execution, the database file is resident on both machines, so the mobile phone only needs to transfer the face image file taken from the camera on the phone. The transfer of the face image adds little overhead in this case, and the collaborative execution performs closely to the execution on the server.

4. Probabilistically guaranteed communication

Reliable communication between mobile devices and server plays an important role in the framework. Due to the dynamics of wireless network and communication networks, we designed a probabilistic-guaranteed connection via virtual devices. In our framework, the reliable communication is realized with virtualization technology. The rationales

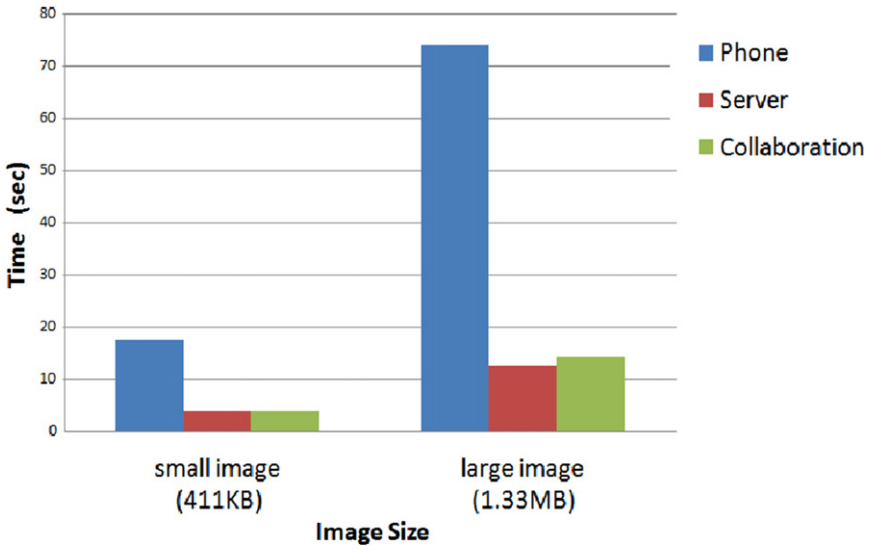

Fig. 8. Performance comparison with a face recognition application.

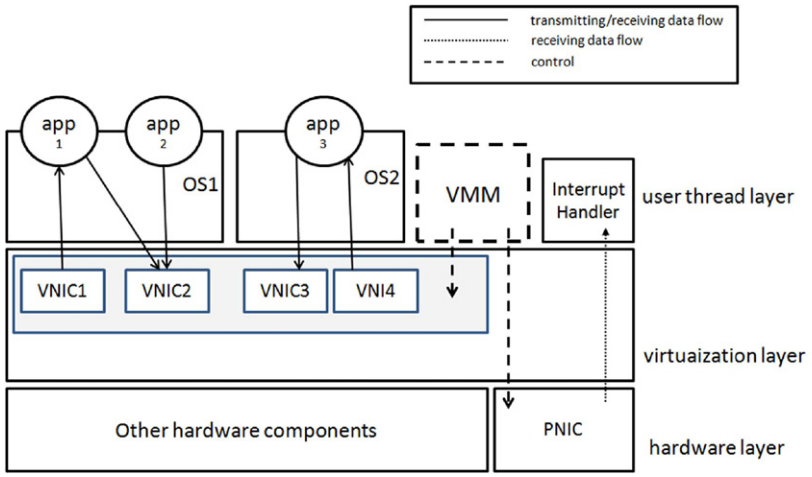

Fig. 9. Architecture for virtual network devices.

are twofold: one is to integrate different QoS services in a virtualized execution framework and the other one is that different types of communication, even in one application, have different QoS requirements. For example, control messages for application migration should be transmitted without fault but usually consist of small amount of data; a video/data streaming transmission can tolerate certain data loss but consists of great amounts of data. To meet all of these different QoS requirements is not trivial. To provide better compatibility and programming interface, we implement the QoS controlling mechanism in a virtualization layer. Therefore we could manage all the QoS requirements using a bandwidth management mechanism.

Fig. 9 illustrates the operation of virtual network devices. The user thread layer provides services to users or manages hardware resource in this system. The threads refer to the programs providing services to users such as OS and stand-alone programs. The user thread layer includes all the user threads. For transmitting, we provide a network bandwidth allocation mechanism according to the bandwidth requirements from services. Due to the dynamics of networking environment, the QoS framework for transmitting provides a bandwidth guarantee with probability. For receiving, we provide a resource protection mechanism to bound the maximum data receiving size for services. To verify and evaluate our framework, we conduct extensive experiments in this work.

Fig. 10 shows one of our evaluation results with three virtual network devices for transmitting. VNIC1 has highest priority and requests for 600 kb/s, VNIC2 has second highest priority and requests for 100 kb/s, and VNIC3 has the lowest priority and requests for best efforts. During the experiment, the available bandwidth to the host changes over time. The result shows that VNIC1 always has the highest probability for transmitting at its requested QoS. When the available bandwidth is no less than 600 kb/s, VNIC1 can transmit at its QoS level. If the available bandwidth is less than its requested bandwidth, it

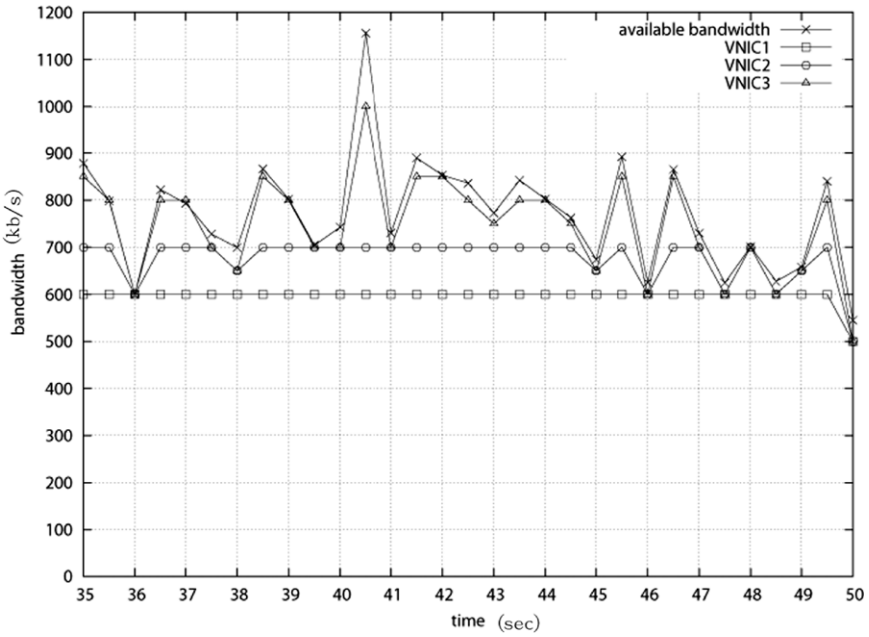

Fig. 10. Probabilistic QoS guarantee by virtual network devices for transmitting.

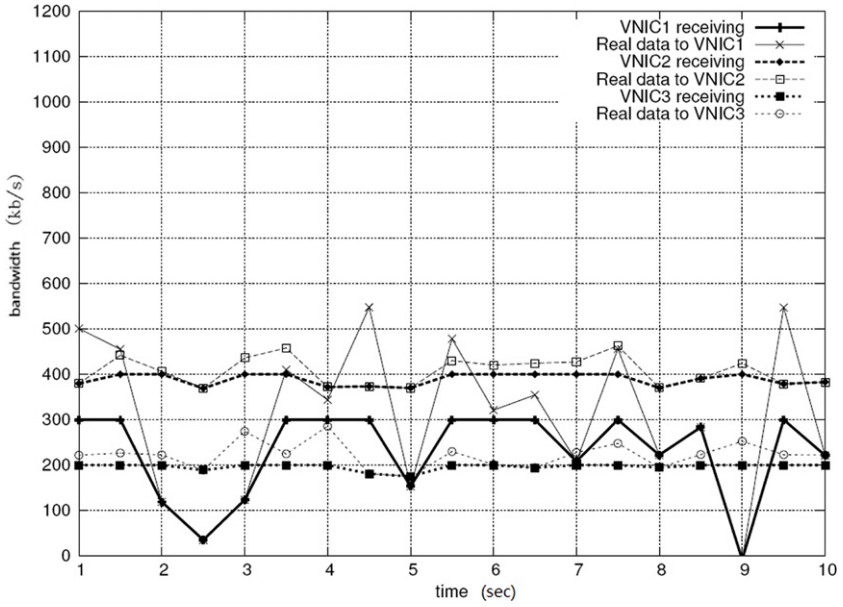

Fig. 11. Probabilistic QoS guarantee by virtual network devices for receiving.

will transmit at lower bandwidth. VNIC3 can only transmit when there is remaining bandwidth. With this mechanism, the application and operating system can be notified if a communication channel can be transmitted at its requested QoS level. Fig. 11 shows the results with three VNIC devices for receiving. The maximum receiving data rates are 300 kb/s for VNIC1, 400 kb/s for VNIC2 and 200 kb/s for VNIC3. For each VNIC, the actual data rates arriving at the VNIC and the forwarded data rates after the QoS control are shown. Our QoS mechanism drops data packets when the maximum receiving rate is exceeded for each VNIC. Note that, to evaluate the capability of bandwidth guarantee, the available bandwidth of the network changes from time to time. As shown in Figs. 10 and 11, the available bandwidth ranges from 500 to 1200 kpbs. In our experiments, VNIC1 has the highest priority. The results show that VNIC1 always transmits at its desired bandwidth when available. The data size on each virtual network interface has no impact on the results because the protocol aims to allow priority data to be transmitted using this approach and the bandwidth is guaranteed in a moving time window.

From the experimental results, we show that our framework could guarantee the resource allocation with probability for transmitting and resource protection for receiving. We also add some extra run-time monitoring mechanism to get better performance. The virtual network devices will lay out the foundation for reliable communication channels in the proposed work. When it is necessary to transmit a message with high robustness, a virtual network device for low bandwidth with high probability will be used.

5. Conclusions

In this paper, we proposed a framework to execute mobile applications in a cloud-based virtualized execution environment controlled by mobile applications and users, with encryption and isolation to protect against eavesdropping from cloud providers. Compared to a conventional scheme, our approach does not require the developers to redesign their applications, but offers the opportunities for the users to migrate their applications from one mobile device to another quickly. More importantly, the user is in control of the application deployment and migration, so the risk of leaking data to application service providers are saved.

We have discussed many practical issues and proposed techniques to address the issues in this paper. At the application level, the agents in our framework collaborate to support application execution in the cloud with techniques, including checkpointing, event recording, event replay, application migration, file synchronization, etc. On the system level, we also design the workload migration framework so that the system has better flexibility to allocate workload on local processing elements or virtual processing elements on cloud servers, depending on network connectivity and core utilization. As a result, the computation resources can be better utilized.

Our framework is still a work in progress. In the future we will continue to improve the framework to deal with the issues outlined in this paper and explore the other related areas. In particular, we are looking into innovative programming models and application programming interfaces for developing collaborative pervasive applications. It is our hope that our framework will provide researchers, users and developers with a versatile environment and insights to make use of mobile cloud computing technologies.

References

[1] B.-G. Chun, P. Maniatis, Augmented smartphone applications through clone cloud execution, in: Proceedings of the 12th Workshop on Hot Topics in Operating Systems, USENIX Association, 2009, pp. 8–8.
[2] A. Rudenko, P. Reiher, G.J. Popek, G.H. Kuenning, Saving portable computer battery power through remote process execution, SIGMOBILE Mob. Comput. Commun. Rev. 2 (1998) 19–26.
[3] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: a Berkeley view of cloud computing, Technical Report, EECS Department, University of California, Berkeley, 2009.
[4] B.-G. Chun, P. Maniatis, Dynamically partitioning applications between weak devices and clouds, in: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond, MCS '10, ACM, 2010, pp. 7:1–7:5.
[5] S. Osman, D. Subhraveti, G. Su, J. Nieh, The design and implementation of zap: a system for migrating computing environments, in: Proceedings of the 5th ACM Symposium on Operating System Design and Implementation, OSDI-02, Operating Systems Review, ACM Press, 2002, pp. 361–376.
[6] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D.R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D.J. Farber, M. Kozuch, C. Helfrich, P. Nath, H.A. Lagar-Cavilla, Pervasive personal computing in an internet suspend/resume system, IEEE Internet Comput. 11 (2007) 16–25.
[7] Android Developers website. [online]. Available: http://developer.android.com/, 15-03-2011.
[8] R.T. Kouzes, J.D. Myers, W.A. Wulf, Collaboratories: doing science on the internet, IEEE Computer 29 (1996) 40–46.
[9] 100,000 Android Applications Submitted To Date, AndroLib Claims, [online]. Available: http://techcrunch.com/2010/07/30/android-market-100000/, 22-02-2011.
[10] J. Flinn, D. Narayanan, M. Satyanarayanan, Self-tuned remote execution for pervasive computing, in: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, IEEE Computer Society, 2001, pp. 61–66.
[11] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, H.-I. Yang, The case for cyber foraging, in: Proceedings of the 10th workshop on ACM SIGOPS European workshop, EW 10, ACM, 2002, pp. 87–92.
[12] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, in: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys'10, ACM, 2010, pp. 49–62.
[13] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield, Live migration of virtual machines, in: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation—vol. 2, NSDI'05, USENIX Association, 2005, pp. 273–286.
[14] J. Flinn, Z.M. Mao, Can deterministic replay be an enabling tool for mobile computing?, in: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile'11, ACM, 2011.
[15] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, P.M. Chen, Revirt: enabling intrusion analysis through virtual-machine logging and replay, SIGOPS Oper. Syst. Rev. 36 (2002) 211–224.
[16] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, V. Inc, Retrace: collecting execution trace with virtual machine deterministic replay, in: Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, vol. 3.
[17] A. Surie, H.A. Lagar-Cavilla, E. de Lara, M. Satyanarayanan, Low-bandwidth vm migration via opportunistic replay, in: Proceedings of the 9th Workshop on Mobile Computing Systems and Applications, HotMobile'08, ACM, 2008, pp. 74–79.
[18] P.E. Sevinç, M. Strasser, D. Basin, Securing the distribution and storage of secrets with trusted platform modules, in: Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 International Conference on Information Security Theory and Practices: Smart Cards, Mobile and Ubiquitous Computing Systems, WISTP'07, Springer-Verlag, 2007, pp. 53–66.
[19] T. Garfinkel, M. Rosenblum, When virtual is harder than real: security challenges in virtual machine based computing environments, in: Proceedings of the 10th Conference on Hot Topics in Operating Systems, vol. 10, USENIX Association, 2005, p. 20.
[20] J. Smith, R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann Publishers Inc., 2005.
[21] C.-C. Chen, S.-H. Hung, C.-P. Lee, Protection of buffer overflow attacks via dynamic binary translation, in: International Conference on Reliable and Autonomous Computational Science, Springer-Basel, 2010, pp. 305–316.
[22] L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala, Rsvp: a new resource reservation protocol, IEEE Commun. Mag. 40 (2002) 116–127.
[23] I. Mahadevan, K.M. Sivalingam, Architecture and experimental results for quality of service in mobile networks using rsvp and cbq, ACM/Baltzer Wirel. Netw. J. 6 (2000) 221–234.

[24] A.K. Talukdar, B.R. Badrinath, A. Acharya, Mrsvp: a resource reservation protocol for an integrated services network with mobile hosts, Wirel. Netw. 7 (2001) 5–19.
[25] D. de Niz, R. Rajkumar, Chocolate: a reservation-based real-time java environment on Windows/NT, in: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), RTAS'00, IEEE Computer Society, 2000, p. 266.
[26] RealVNC website. [online]. Available: http://www.realvnc.com/, 25-03-2011.
[27] QEMU open source processor emulator website. [online]. Available: http://wiki.qemu.org/, 23-04-2011.
[28] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
[29] J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger, Readings in Database Systems, second ed., Morgan Kaufmann Publishers Inc., 1994, 181–208.
[30] android-openvpn-settings website. [online]. Available: http://http://code.google.com/p/android-openvpn-settings//, 26-07-2011.
[31] OpenVPN-Open Source VPN website. [online]. Available: http://openvpn.net/, 23-07-2011.
[32] Google Code Project website. [online]. Available: http://code.google.com/, 08-03-2011.