Software Engineering Institute

Application Virtualization as a Strategy for Cyber Foraging in Resource-Constrained Environments

Dominik Messinger
Grace Lewis

May 2013

TECHNICAL NOTE
CMU/SEI-2013-TN-007

Research, Technology, and System Solutions Program

http://www.sei.cmu.edu

Carnegie Mellon

Copyright 2013 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This report was prepared for the

SEI Administrative Agent
AFLCMC/PZE
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution except as restricted below.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

DM-0000313

Table of Contents

List of Figures

List of Tables

Abstract

Modern mobile devices create new opportunities to interact with their surrounding environment, but their computational power and battery capacity is limited. Code offloading to external servers located in clouds or data centers can help overcome these limitations. However, in hostile environments, it is not possible to guarantee reliable networks, and thus stable cloud accessibility is not available. Cyber foraging is a technique for offloading resource-intensive tasks from mobile devices to resource-rich surrogate machines in close wireless proximity. One type of such surrogate machines is a cloudlet—a generic server that runs one or more virtual machines (VMs) located in single-hop distance to the mobile device. Cloudlet-based cyber foraging can compensate for missing cloud access in the context of hostile environments. One strategy for cloudlet provisioning is VM synthesis. Unfortunately, it is time consuming and battery draining due to large file transfers. This technical note explores application virtualization as a more lightweight alternative to VM synthesis for cloudlet provisioning. A corresponding implementation is presented and evaluated. A quantitative analysis describes performance results in terms of time and energy consumption; a qualitative analysis compares implementation characteristics to VM synthesis. The evaluation shows that application virtualization is a valid strategy for cyber foraging in hostile environments.

1 Introduction

1.1 Background and Motivation

Mobile computing is now commonplace in our society, and its impact on our everyday lives is steadily growing. With smartphones having had their commercial breakthrough in recent years [Mobithink 2011, Infographic 2012], a whole ecosystem of applications has evolved that today shapes the way its users interact with the world surrounding them. Context-aware services such as localization help to find nearby venues or find where friends are currently located. Built-in cameras enable users to share visual impressions immediately as well as to scan and process information in the direct environment.

Ubiquitous computing began with a vision: it would provide information that exceeded our natural recognition capabilities and would thereby augment human perception. That vision is already a reality.

However, there are still limitations to mobile devices because they tend to fail to meet the needs for performing resource-intensive tasks due to their restricted battery capacity and computing power. Nonetheless, resource-intensive applications such as natural language processing, face and speech recognition, or decision making are among the most desired services for mobile devices [Satyanarayanan 2009]. To deal with the issue of resource limitation, techniques have been developed that offer mobile devices access to increasingly powerful external computing facilities that overtake the burden of resource-intensive computations. Most notably, cloud computing provides today's mobile devices with resources that extend the mobile device's capabilities.

Popular commercial cloud-connected mobile applications include the natural language-processing software *Siri*, which must access the Apple Cloud, or *Google Goggles* communicating with Google servers, in order to provide image recognition [Apple 2012, Google 2012a]. To use cloud resources, the mobile device must connect to the internet in order to establish a connection to the cloud services. While this is an appropriate solution for many use cases, certain shortcomings to the use of clouds still remain.

1.1.1 WAN Latency as a Limitation to Cloud Resources

One main drawback of relying on cloud resources is latency. Latency is determined by the distance between the mobile device and the cloud, network bandwidth, and the processing time on client and server side as well as within the network. Latency increases due to packet drops and various software or hardware layers, such as routing mechanisms, congestion avoidance algorithms, integrity checks, or security layers. Satyanarayanan and colleagues stated that wide area network (WAN) latency is not going to improve because modern networking research typically focuses on issues such as security and manageability [Satyanarayanan 2009]. Solutions to these issues often lead to an increase in the additional overhead per transmitted packet. Providing low latency services is essential for fast-responding applications, for example, augmented reality software that must process and display information in real-time. Therefore, data centers must be in close proximity, depending on the type of service that the mobile device demands. For example, highly responsive augmented reality applications require reasonably high frame rates in order to provide an acceptable usage experience [Satyanarayanan 2009, p. 17]. As long as services are bound to predetermined servers, these must be close enough to the mobile device to provide acceptable performance.

1.1.2 Resource-Constrained Environments Without WAN Access

The assumption of good internet access, which satisfies the mobile device's demand on bandwidth and reliability, may be incorrect in some cases. Cloud computing is infeasible in unreliable networks, which occur in theaters of military operations and in the context of disaster recovery. An example of a resource-constrained environment with such conditions has been described as a *hostile environment* by Ha and colleagues [Ha 2011]. Hostile environments, in contrast to areas with well-established network infrastructures, cannot assume connectivity to wide-area networks. Wide-area networks may be unavailable because of serious infrastructure problems, for example, as a consequence of earthquakes or war actions. Opponents who intrude onto the network and carry out attacks may also compromise such networks. Considering such cyber war attacks, Ha and colleagues assume that even the internet may become one day a hostile environment, as they define it above [Ha 2011]. The Department of Defense has shown a strong interest in equipping soldiers with handheld devices to enhance their operational abilities [Morris 2011]. It cannot risk relying on unsafe networks but needs access to a stable infrastructure that can meet the advanced safety and security demands of the military.

1.1.3 Cyber Foraging and VM Synthesis in Hostile Environments

Ha and colleagues propose a solution for code offloading, that is, transferring resource-intensive tasks to stronger external machines, in hostile environments [Ha 2011]. It utilizes cyber foraging techniques in single-hop networks and virtualization technology. The term *cyber foraging*—first introduced by Satyanarayanan—describes the technique of code offloading to nearby surrogate machines, called *cloudlets* [Satyanarayanan 2001]. These cloudlets are in close proximity to the mobile device and are accessible via a single-hop network. Such a setup differs from one that relies on distant clouds, and it does not suffer from the previously mentioned shortcomings for cloud computing in the context of hostile environments. Single-hop networks guarantee low-latency connections and are generally not as vulnerable to cyber-attacks as are WANs [Ha 2011, p. 4].

In order to enable cloudlet-based cyber foraging, a mobile application is divided into a client running on the mobile device and a server running on the cloudlet. The server must be deployed on the cloudlet before it can become accessible to the mobile client. This deployment is accomplished via *VM synthesis*: the cloudlet holds Virtual Machine (VM) images and receives *VM overlays* that enable the cloudlet to reconstruct a complete VM that includes the application server. A VM overlay is the binary difference between a base VM snapshot and a snapshot of a base VM clone after installation of the server. After receiving the overlay, the cloudlet merges this delta and the base VM; this results in a complete system that is ready for execution. A second, revised solution transfers two overlays: the disk image overlay and the memory snapshot overlay. Providing a snapshot of the memory enables the cloudlet to resume the reconstructed VM from a suspended state rather than conducting a cold start.

Simanta and colleagues present a reference implementation in their report [Simanta 2012]; Figure 1 depicts the process of overlay creation.

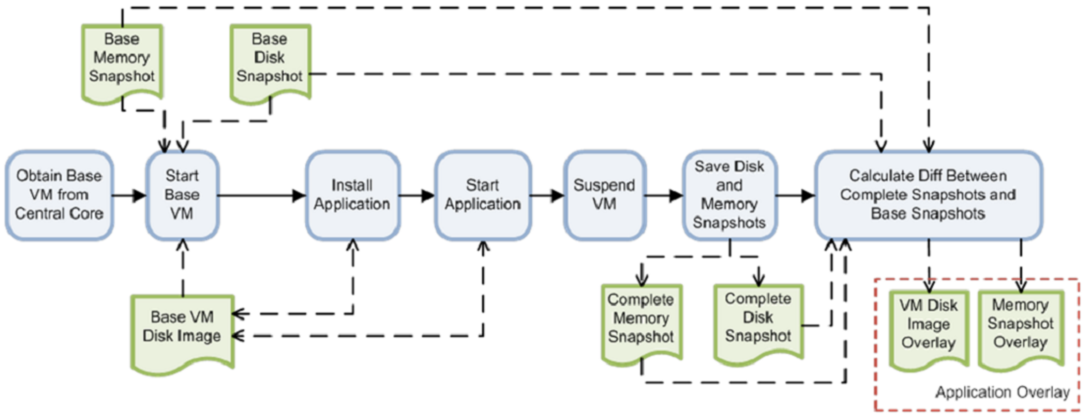

Figure 1: VM Overlay Creation [Simanta 2012, p. 15]

Cyber foraging in combination with VM synthesis offers a simple solution to deal with unreliable networks in hostile environments. Nevertheless, there are shortcomings in terms of performance and flexibility. When the cloudlet does not have access to distant storage of VM overlays, the mobile device has to be responsible for overlay transmission. An overlay is calculated as the straight binary difference between a VM before application install and after application install. Consequently, overlays tend to be significantly larger than the actual application server because the binary difference includes information that is irrelevant to the application. Transfer time between mobile and cloudlet, as well as battery consumption, increase proportionally to the overlay size [Simanta 2012, p 19]. Regarding flexibility, VM synthesis requires matching a VM overlay with the same base VM that was used during the creation of the overlay. Therefore, any updates to the base VM require recreation of the overlay because the VM synthesis process is based on the binary difference.

1.1.4 Goal and Structure of this Technical Note

The purpose of this technical note is to explore the applicability of application virtualization as a strategy for cyber foraging in resource-constrained environments. Application virtualization emulates operating system services for applications. This approach is more lightweight than VM synthesis, whose virtualization technique emulates hardware for complete operating systems. In this technical note, we describe the implementation of a cyber-foraging framework that utilizes application virtualization to provision cloudlets with application servers.

This technical note begins with an introduction to cyber foraging and cloudlets, and continues with a discussion of different techniques for application deployment. We introduce application virtualization, then present and evaluate the implementation. The evaluation includes a comparison with VM synthesis regarding its suitability for operation in hostile environments. Finally, we identify the limitations that inspire topics for future work.

2 Cyber Foraging

2.1 Concept

Mobile devices suffer from resource constraints that restrict their computing capabilities for resource-intensive tasks. Although over time mobile devices are gaining more computing power, they are unlikely to become as powerful as static machines, such as desktops and servers. The requirements for mobile devices—such as low weight, small size, long battery life, and operation at skin-friendly temperatures—contradict the assembly of the best available hardware. Unfortunately, resource-intensive tasks such as natural language processing, image and speech recognition, and decision making are among the most desired applications for mobile computing [Satyanarayanan 2009].

Cyber foraging, as first introduced by Satyanarayanan [Satyanarayanan 2001], is a technique to enable resource-poor, mobile devices to leverage external computing power. Therefore, it circumvents the outlined resource restrictions. A mobile device offloads code to a so-called *surrogate* [Satyanarayanan 2001] machine, taking advantage of a more powerful hardware infrastructure. This surrogate executes the code and returns the computational result to its client.

2.2 Scenario

Consider the following scenario for cyber foraging where the surrogate is part of a cloud. Figure 2 presents an illustration.

> *Susie works as a security guard at the entrance of a football stadium. The next ticket holder in line approaches and Susie's colleague searches him for prohibited items such as fireworks. In the meantime, Susie needs to find out if he is on the blacklist and therefore not permitted to enter the stadium. She takes her smartphone, points the camera at the ticket holder and starts the face recognition application. The application connects to the cloud and transmits the pictures from the camera. On the cloud, the face recognition server looks for a match in the photo database of known hooligans. When it finds no match, Susie lets the ticket holder pass, wishing him a great time and good luck to his team.*

Processing the face recognition locally on Susie's phone would be too slow for her demands and would probably drain her battery after several uses. Cyber foraging enables her to extend her phone's computing power, thus empowering her to reduce the overall risk of hooligan riots within the stadium.

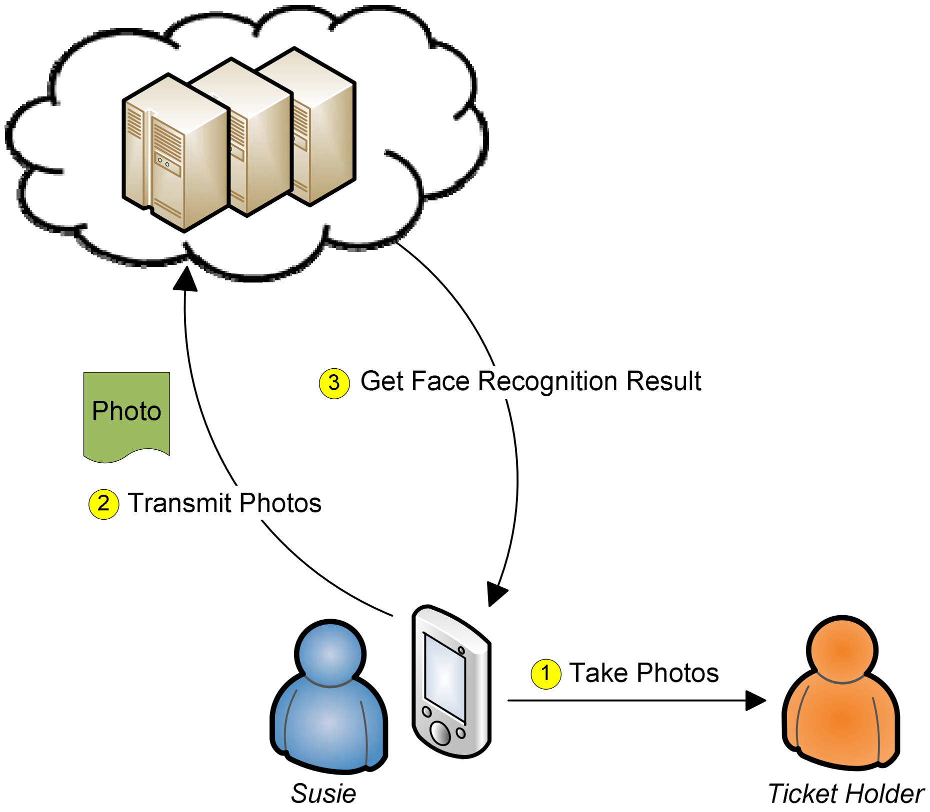

Figure 2: Cyber-Foraging Scenario "Stadium Security"

2.3 Cyber-Foraging Strategies

In order to relieve the mobile client, the surrogate machine must be capable of running offloaded tasks. It is therefore necessary to install on the surrogate a software item that serves this need. This software item may range from a standard Web service to a complex software system that is specialized for code offload. We refer to approaches that differ in terms of deployment effort as *cyber-foraging strategies*. This section provides some concrete examples of cyber-foraging strategies.

2.3.1 Pre-Installed Applications

In the simplest case, the surrogate is ready for execution and no code must be offloaded. This is therefore the lightest form of deployment. Examples are Web services that are already installed, or components that support remote procedure calls. Such software is accessible through an interface that is known to the client. This interface is typically defined through an interface description language, for example, *WSDL* [Chinnici 2007], and *CORBA IDL* [OMG 2011].

A real-world cyber-foraging example is a smartphone app that uses the Google Maps API Web Services [Google 2012b] for computing the shortest distance between two given locations.

2.3.2 Mobile Code

Another deployment approach has the surrogate execute portions of code that are offloaded from the client. This approach is different from that described in Section 2.3.1, as it involves not only transmission of data—that is, service identifier plus arguments— but also transmission of code from the client to the surrogate.

Partitioning code into local and remote can be done manually or automatically; either the developer marks code portions for remote execution or an automatic tool uses advanced code inspection or profiling to identify remote code. On the remote side, the surrogate provides a runtime environment that executes the offloaded code. Depending on what the runtime environment expects, the remote code could be source code, bytecode, or machine code.

Several cyber foraging strategies have been proposed that follow the mobile code approach. *Spectra* [Flinn 2001] and its successor *Chroma* [Balan 2002] require the developer to modify the source to identify code for remote execution. The developer can influence how the remote execution is performed by setting quality requirements and, in the case of Chroma, by defining *tactics* that declare alternatives for sequential or parallel operation of remote procedures [Balan 2003, 2007]. Similar solutions are *Scavenger* [Kristensen 2010] and *MAUI* [Cuervo 2010], which support code annotations to mark procedures for remote execution.

Designed to let mobile devices have the benefit of a cloud, *CloneCloud* offers automatic partitioning at the thread level by finding the migration profile with the least migration cost [Chun 2011]. No source code modification is necessary. During execution, the control flow migrates between the mobile device and the cloud. The cloud hosts a device clone that resides within a VM; this clone serves as the offload site. An older automatic partitioner that is unrelated to mobile computing is *Coign* [Hunt 1999]. Restricted to the Microsoft Component Object Model (*COM*) [Microsoft 1993], Coign identifies components for remote placement by intercepting inter-component communication.

2.3.3 Application Deployment

Another way of implementing cyber foraging is to deploy a self-contained application on the surrogate at runtime. After the installation is finished, the mobile client can then communicate with the application to execute the resource-intensive code. Application deployment strategies do not require complex middleware such as the runtime environments and code partitioning tools described in 2.3.2. Instead of working on fine-grained separation into remote and local code, the developer implements a client-server architecture, whereby the server is the dedicated part for execution on the surrogate machine. An additional advantage of this approach is that the server implementation does not have to be code that runs on the mobile device, therefore creating the potential for much powerful applications.

Goyal and Carter introduced a cyber-foraging strategy that implements application deployment [Goyal 2004]. The mobile device triggers the surrogate to download the requested application from the internet and install it within a VM, where it is isolated from other applications.

2.3.4 Virtual Machine Deployment

The artifact of virtual machine deployment is a complete VM that contains the application. Unlike application deployment, the deployment process is not the installation of an application in a VM, but rather deployment of the entire VM on the surrogate.

Satyanarayanan and colleagues use a technique called *VM synthesis* to provision the surrogate machine [Satyanarayanan 2009]. The binary difference between VM snapshots that are taken before and after application installation is computed offline and sent over to the surrogate machine at runtime. The surrogate, which stores the original *base VM* without the installation, can then restore the application-ready VM. Wolbach describes this technique [Wolbach 2008].

2.4 Application Virtualization as a Cyber-Foraging Strategy

The work that we describe in this technical note uses application virtualization as a cyber-foraging strategy and compares it with the VM synthesis strategy. Application virtualization belongs to the category of application deployment (see Section 2.3.3) and VM synthesis to the category of virtual machine deployment (see Section 2.3.4).

3 Cloudlets

3.1 Concept

Instead of relying on distant computing clusters such as clouds, Satyanarayanan and colleagues propose the use of *cloudlets*. A cloudlet is a computer or computer cluster that serves as a code-offloading site for nearby mobile devices [Satyanarayanan 2009]. The close one-hop proximity to such a cyber-foraging surrogate avoids possible high latencies. Cloudlets leverage the benefits of local-area networks such as low-latency, high bandwidth and less vulnerability to cyber-attacks compared to wide-area networks such as the internet [Satyanarayanan 2009 p. 15, Ha 2011, p. 4]. In contrast to clouds, cloudlets are decentralized machines and each cloudlet is managed separately. The deployment and maintenance of cloudlets should follow the principle of simplicity and the cloudlet should not keep any critical state [Satyanarayanan 2009, p. 9].

3.2 Architecture

Ha and colleagues propose a two-level hierarchy with cloudlets at the edge and a cloud at the core as shown in Figure 3 [Ha 2011]. The cloudlets serve as offload elements for mobile devices. Connectivity between the cloudlets and the cloud is only required for provisioning—cloudlets do not depend on the cloud for fulfilling their purpose as surrogates to mobile clients. A cloudlet is considered to be stateless but may cache state to speed up later use. Because no essential state is kept, it takes little effort to install or replace a cloudlet. The motivation for building this architecture was increased computing power and battery life on mobile devices in hostile environments [Ha 2011, p. 8]. In contrast, Satyanarayanan and colleagues assumed [Satyanarayanan 2001] that cloudlets would have permanent internet access [Satyanarayanan 2001, p.14].

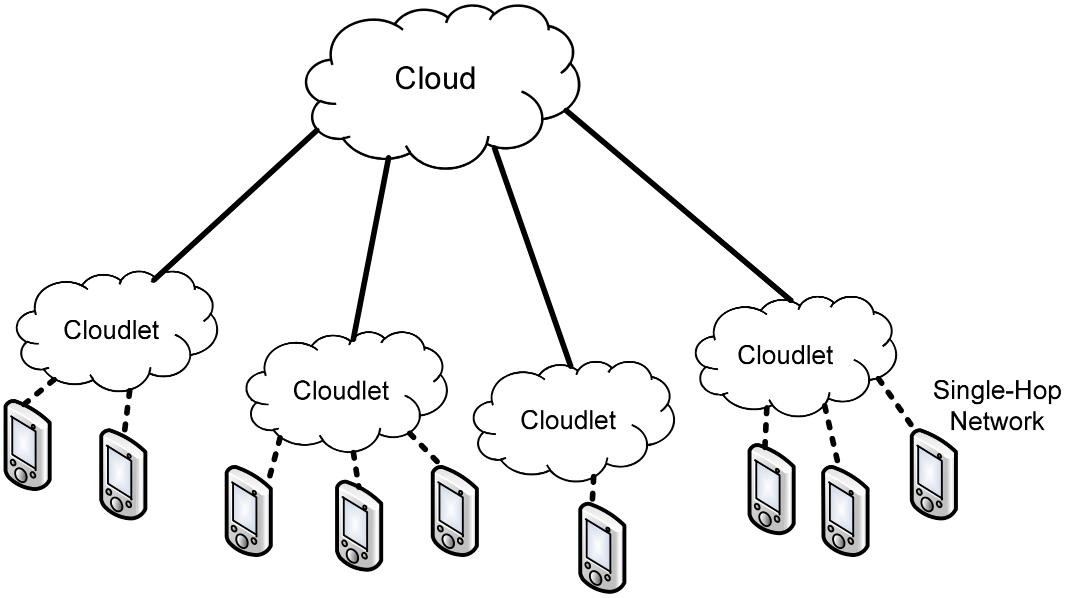

Figure 3: Hierarchical Architecture for Offload to Cloud-Connected Cloudlets Based on Work by Ha and Colleagues [Ha 2011]

3.3 Cloudlet Scenario

The following cyber-foraging scenario is set in a hostile environment and therefore does not rely on the internet. Figure 4 presents an illustration.

> *Susie quit her job as a security guard and now works for an NGO that focuses on disaster recovery. Recently, a massive earthquake occurred, causing a high number of casualties and leaving an entire region in pure chaos. Many houses have been destroyed and the telecommunication infrastructure is down. Susie's task is to go to countryside villages and interview the survivors to get an idea of the total damage in order to better decide how to effectively coordinate the first-responders' efforts. Because Susie does not speak their language, she uses her smartphone's live translation service to communicate. First, she transfers the live translation application to a cloudlet that is installed in her car. Now she can record speech on her phone, send it to the cloudlet for translation, and receive a translated text transcript. Finally, Susie's phone uses text-to-speech functionality to voice the translated sentence in the local language. In the same way, her phone enables her to understand the people that she interviews.*

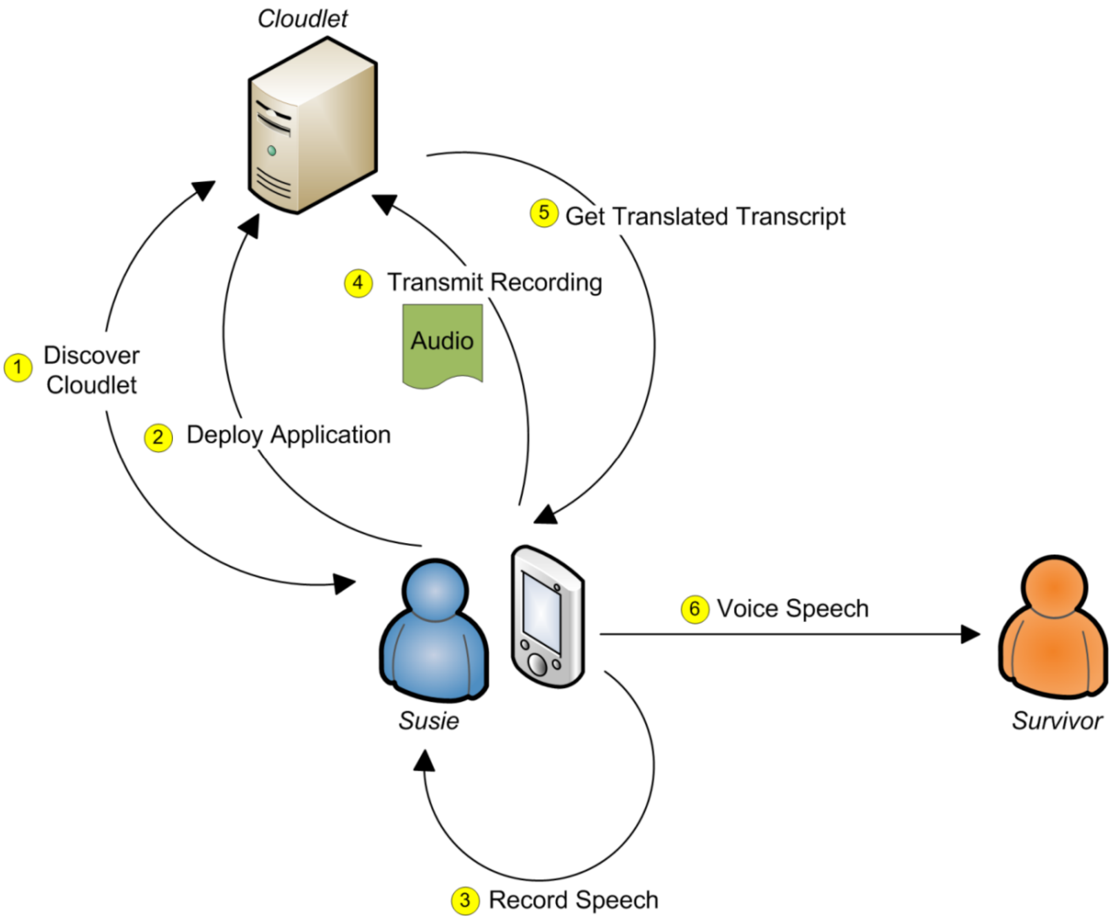

Figure 4: Cloudlet Scenario

3.4 Phases of Cloudlet Interaction

The interaction between a mobile device and a cloudlet can be divided into four phases that describe the necessary steps for cyber foraging. Figure 5 depicts these phases. Because a cloudlet offers its platform and not pre-installed services, these phases include application deployment.

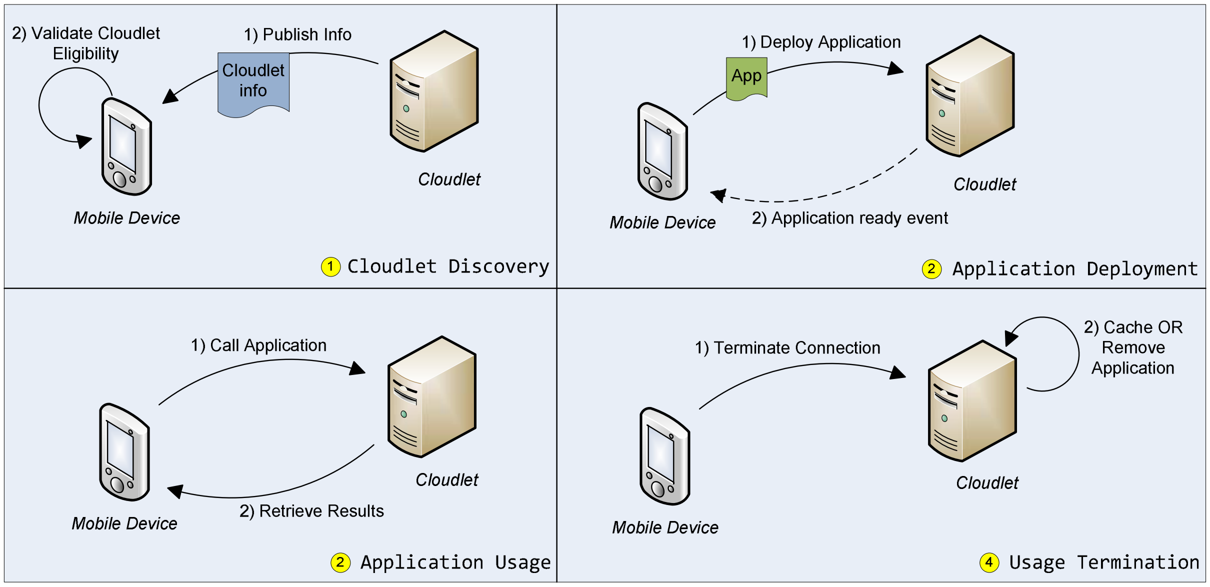

Figure 5: Phases of Cloudlet Interaction

1. Cloudlet Discovery: The mobile device discovers nearby cloudlets and selects the cloudlet that best meets its requirements. This means that each cloudlet must publish information about itself, which is retrievable through a discovery mechanism.
2. Application Deployment: After the mobile device has selected an appropriate cloudlet, it must deploy the application that it would like to have executed remotely. The application may already be deployed; if not, the mobile device must be able to deploy it.
3. Application Usage: As soon as the application is deployed on the cloudlet, the mobile device starts to interact with it in order to accomplish its tasks.
4. Usage Termination: When the mobile device has no further need of the application on the cloudlet, it terminates its connection. The application may be cached for possible later use or may be removed instead.

3.5 Cloudlet Requirements Analysis

To deliver the benefits of being easy to deploy, multi-purpose, and transient, cloudlets must meet the following functional and quality attribute requirements:

3.5.1 Functional Requirements

1. A mobile device must discover all cloudlets that are in the same wireless network (same subnet).
2. Each cloudlet must publish information about its characteristics, thus allowing the mobile device to find the most suitable cloudlet for the application that it wishes to offload.

3. A mobile device must select a suitable cloudlet for application deployment if available; otherwise, it should inform the user that there are no cloudlets available.
4. Each application must have associated information about its required cloudlet characteristics.
5. If a cloudlet's characteristics match the application's requirements, the cloudlet must guarantee correct installation and operation of the application.
6. During the interaction with the mobile device, especially during application deployment, the cloudlet must not rely on external resources such as machines or data storage connected to the internet.
7. A cloudlet must be capable of serving multiple mobile clients at a time.
8. Serving one mobile client must not affect the correct operation for serving other mobile clients.
9. A cloudlet must be able to remove a client application completely after use (if required).

3.5.2 Quality Attribute Requirements

1. The application deployment should be reasonably fast and take less than 2 seconds per MB of application package size.
2. The interaction with the cloudlet should be battery efficient and cost the mobile device less than 3 Joule per MB of application package size.
3. A cloudlet should be generic, that is, able to host a variety of applications, and not be limited to a few applications.
4. A cloudlet should allow upgrades and patches without losing its ability to host particular applications.
5. Deploying a cloudlet should be simple and require nothing more than setting up a standard runtime environment, installing an application, and adjusting a configuration file.
6. Making an application ready to use on a cloudlet should be simple and require nothing more than usage of an automated tool followed by further manual adjustments to the tool's output.
7. The mobile device should expect the cloudlet only to meet those requirements that are essential for a correct execution of the application.

4 Application Deployment

Before a cloudlet can execute an application on behalf of a mobile device, it must be provisioned with the application that will serve the mobile client's requests—the code offload process. This section discusses several general approaches for how to deploy an application on a remote machine. Because a cloudlet does not guarantee connection to the internet or to other external sources, the mobile client must provide all data necessary for deployment. An application should be deployable in as many different environments as possible, to increase the chance of discovering a cloudlet that satisfies the mobile client's needs. We thus aim to port an application from its original environment to another; hereby we distinguish between the *source system* and *target system*. The desired solution would be a mechanism that maximizes an application's portability, thus minimizing the coupling between source and target system.

4.1 Limitations to Portability

There are certain constraints that must be imposed if an application is to be ported to another system.

4.1.1 Instruction Set Architecture

The instruction set architecture on the target machine must be compatible with the one for which the application has been designed. For example, a 64-bit binary that has been compiled for x86-64 architectures cannot run on 32-bit x86 architectures. Because applications often have third-party dependencies that are either 32-bit or 64-bit, the target system must provide the same instruction set architecture.

4.1.2 Hardware Dependencies

If the application relies on pre-defined hardware, such as a special sensor or a special GPU, the target system must provide this hardware. In some cases, such hardware may be emulated instead to make the application work on the target system.

4.1.3 Software Dependencies

Software often depends on specific versions of other software. The concept of shared libraries allows software modules to be used by more than one program; a shared library must exist only once on the hard disk and can be shared in memory by different processes. When an application depends on a shared library, the library is linked at either load time or runtime. Therefore, the library must exist on the target system; otherwise, the linker will eventually fail because it cannot resolve a symbol, resulting in a broken application.

It is important to keep in mind that the cloudlet cannot access the internet for downloading missing software. Consequently, the application to be deployed on the cloudlet must include all software dependencies. Because a software module may be missing on one target system but may exist on another, it is difficult to determine which dependencies must be delivered with the application. For example, different Linux distributions have different sets of installed libraries. There is a large variety of Linux distributions and the user may re-compile the Linux kernel, including only necessary libraries, or may continuously add or remove libraries depending on system changes. Relying on the library set of a specific distribution would drastically reduce the number of valid target systems. This contradicts the goal of minimizing the mobile device's assumptions about a cloudlet (see Section 3.5.2).

4.1.4 Dependency Conflicts

If a shared library is updated, a conflict may arise because other software modules are no longer compatible with the new version of the library. This issue can be resolved by allowing multiple library versions to be installed side-by-side so that modules can still use outdated versions. Another type of conflict arises if a software module implements functionality that risks breaking other software. To this effect, Linux packages such as Debian or Red Hat contain metadata that includes name dependencies of possible conflicts [Jackson 2012, 2005].

These two types of dependency conflicts are normally avoided by using a package manager that maintains the system's state of installed libraries. It can fetch dependent libraries and remove conflicting libraries, or prohibit the installation of packages that would cause conflicts. The package manager is therefore a system tool that can change the operating system's state significantly. For this reason, it is not suitable for application deployment on a cloudlet. Applications must not alter the target system in a way that hinders deployment of other applications. Furthermore, the cloudlet's ability to install applications should not rely on any special pre-installed libraries. To some extent, this does not include system libraries and other basic libraries normally present on the target system.

4.2 Source Code Versus Binary File Transmission

Transmitting source code instead of executable binary files seems to increase portability at first glance because it allows direct compiling for the target system. However, this benefit is achievable only if all source code dependencies are available on the target system. Otherwise, already compiled dependencies narrow the range of possible target systems, regardless of the main application's sources. The assumption of source code availability is invalid for use of non-open-source software. In this case, shipping the application as a binary does not limit portability further and, in addition, avoids the time-consuming compilation on the target system.

4.3 Packaging Dependencies

In keeping with the arguments in the previous sections, the file that the mobile device transfers to the cloudlet must contain the application with all of its dependencies. Installing the application must not alter the target system more than is necessary. The following sections discuss alternatives for how to create this file, which we call a *package* to indicate that it is an archive consisting of multiple files.

4.3.1 Remote Install

Remote install requires transferring the application along with the software packages that it depends on, and then installing these packages on the target system. This is a straightforward approach, but it has some drawbacks. As Section 4.1.4 describes, conflicts may arise when installing software packages. These may break the correct execution of the application. Furthermore, installation alters the target system; it may even remove its capability to run or install other applications. The installation packages must fit the target system, thus making strong assumptions about the cloudlet and its configuration. Another drawback is the additional time that it takes to install new packages on the target system. Overall, remote install strongly contradicts the goal of not altering the cloudlet state.

4.3.2 Library Packaging

Instead of including installation packages in the transfer, it is possible to send instead a set of required libraries and manipulate the target system's linker into preferring those over the libraries that are already installed on the target system. This allows porting of the packaged application to the same operating system distribution that corresponds to the source system. However, successfully porting from one distribution to a different distribution cannot be guaranteed. As an example, during this work, we tried to run an application that had been packaged for Ubuntu 12.04 on Ubuntu 10.04. When using the native standard C library of the target system—Ubuntu 10.04—the execution failed because the application's dependencies required a different library version. When using instead the correct version of the standard C library from the Ubuntu 12.04 source system, the execution resulted in a segmentation fault and was terminated.

This example shows that library packaging, like remote install, prevents porting applications across distribution boundaries. However, it is desirable for a cloudlet to be able to run an application from the mobile client, even if the target system that is offered by the cloudlet differs from the source system's distribution where the application has originally been packaged.

4.3.3 Static Linking

Static linking allows for including the compiled code and all library dependencies in one binary. When the linking succeeds, the execution will not fail during load time because of missing dependencies. However, to link successfully, the order in which the linker receives the library arguments is important. For example, if A depends on B (i.e., A uses symbols that are delivered by B), the linker must link A first. Also, if there are cyclic dependencies, static linking will fail. Another issue is the need of static libraries for linking; if only shared libraries are available, then static linking will be impossible. Furthermore, copyright licenses are likely to forbid the inclusion of third-party source code through static linking. Concerning the limitation to specific OS distributions, static linking resembles library packaging. Therefore, in terms of portability, static linking is not a good solution.

4.3.4 Application Virtualization

Application Virtualization, like library packaging, includes all shared libraries. But instead of only changing the location in which the loader searches for libraries, it encapsulates the application in a more extensive way. Application virtualization uses a similar approach to OS virtualization, which is "tricking" the software into interacting with a virtual rather than actual existing environment. While OS virtualization emulates the hardware for a guest OS, application virtualization emulates OS functionality for an application. To accomplish this, a runtime component intercepts all system calls from an application and redirects these to resources inside the virtualized application. Typically, a virtualized application has its own file system, registry (if necessary), and environment. The runtime redirects I/O operations and library calls to files that reside within the virtual application package and performs registry operations on the internal database. The application itself is unmodified and unaware that it is interacting with virtual operating system services. Figure 6 illustrates this concept.

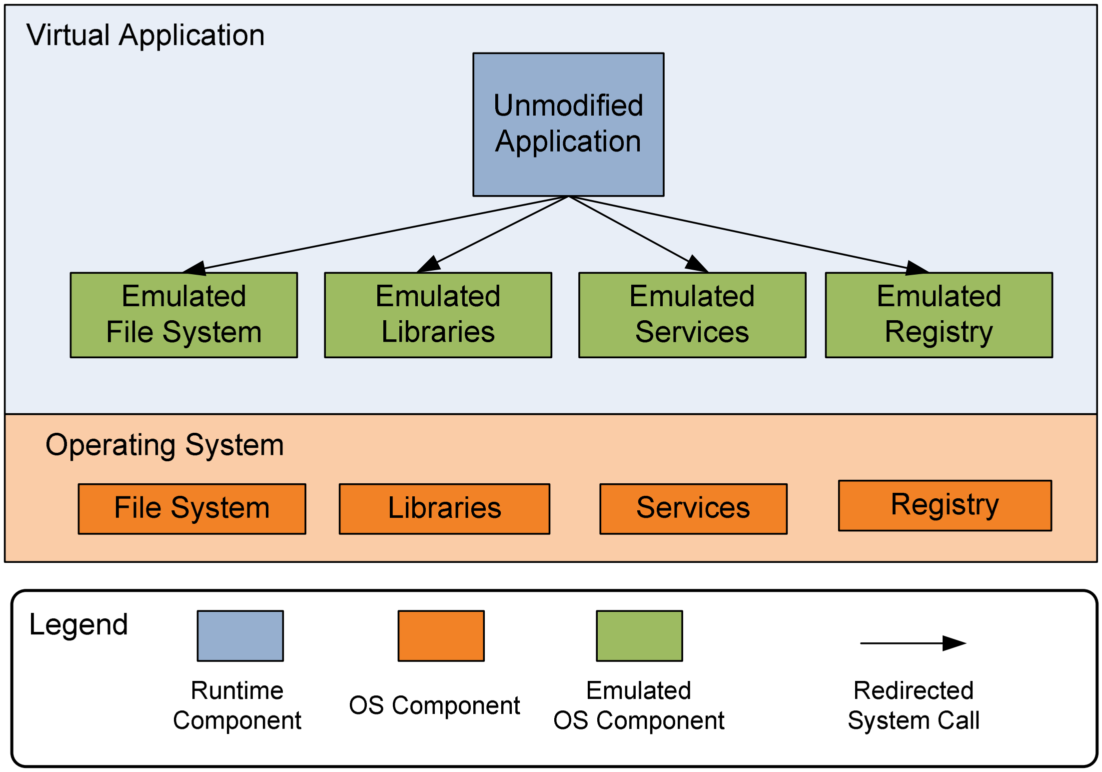

Figure 6: Application Virtualization Through OS Component Emulation and System Call Redirection

Application virtualization is not usable for every type of application. For example, device drivers, because they interact with the hardware directly, cannot be virtualized. It is also difficult to virtualize software that interacts with the OS internals, such as antivirus programs.

A virtualized application generally executes more slowly than its non-virtualized form because every user/kernel mode switch that is caused by a system call results in two additional user/kernel mode switches: the first to change from the kernel to the virtualization runtime, and the second from the virtualization runtime back to the kernel to execute the modified system call. However, application virtualization offers portability to a degree that the approaches we discussed earlier cannot guarantee. Because a virtualized application is partly isolated from the OS, it can be ported across distribution boundaries more easily. The cloudlet architecture implementation in this technical note uses application virtualization tools to package the application for its transfer to the cloudlet. The next section presents the basic design goals of this implementation along with an introduction to the application virtualization tools that we used.

5 Application Virtualization for Cloudlets

5.1 Design Goals

The cloudlet architecture implementation described in this technical note uses application virtualization techniques to support the design goals selected for the implementation: simplicity, generality, and quick response.

Simplicity

> Setting up a cloudlet should be convenient and accomplished in a short time without major changes to the system. Making an application ready for deployment on a cloudlet should be easy and avoid manual overhead. Cloudlet discovery should not require any action from the user. Offloading to the cloudlet (i.e., shipping deployable applications) must be intuitive and simple. The change from deployment phase to the application usage phase must be seamless. The user must not have to worry about low-level communication details such as IP addresses and ports.

Generality

> Packaged applications should be loosely coupled to the operating system so that they can run on many cloudlets. As a result of loose coupling, the cloudlet should allow regular updates and upgrades to the operating system that runs virtual applications without breaking functionality. All applications that are not integrated too deeply into the operating system or specific to special hardware should be eligible for offloading to the cloudlet.

Quick Response

> The time from the user selecting an application for offload and the application to be ready for use (application-ready time) should be reasonably small. The user must be able to track the deployment progress by receiving progress messages from the cloudlet.

Application virtualization can address these goals because it does not require any code modifications and provides a high degree of application portability. The file size of the virtualized application strongly influences the application-ready time. Application virtualization tools package only those dependencies that are necessary for portability, thus keeping the file size small.

5.2 Application Virtualization Tools

We used two tools for creating and executing virtualized applications: one for Linux and one for Windows systems. We selected these tools because they were freely available and are among the most mature non-commercial tools (determined simply by Web presence and adoption claims).

5.2.1 CDE

CDE (short for Code, Data, and Environment) is an application virtualizer for Linux developed by Philip J. Guo and Dawson Engler [Guo 2011]. CDE allows for virtualizing applications by monitoring their execution. Through the *ptrace* system call, the supervising CDE program finds files that have been accessed during execution and packages them [Linux 2013]. The resulting package also contains the environment settings and the CDE runtime environment, which executes the virtual application. The original directory structure that contains the accessed files is mirrored inside the package. Every time the virtualized application tries to access a file, the corresponding system call is intercepted by the CDE runtime, which serves as an additional layer between the original application and the operating system. Instead of accessing the original file path, the path is changed to the corresponding location within the package. In this way, accessed libraries and data are independent from the operating system on which the virtualized application is executed. The authors indicate that “packages created today (in 2011) will run fine on Linux 2.6 distros (distributions) that take place several years in the future” [Guo 2011, p 4].

The package can be configured to allow access to specified file paths outside its sandbox. CDE does not guarantee including all dependencies in the package. In general, every tool that automatically detects dependencies is incapable of finding every dependency. In order to find all files that theoretically are accessible, every possible control path would have to be examined. This is a non-deterministic problem; solving it would require predicting program behavior before execution. Especially for applications that implement a plugin structure and dynamically load libraries during runtime, dependencies are likely to be missed. To address this issue, the CDE packager can be run several times, each time adding files that have been newly accessed by the application. It is also possible to add files manually to the package; this way is preferred way when deeper knowledge about the application exists.

5.2.2 Cameyo

Cameyo is an application virtualizer for Windows. It packages the application and its dependencies into one single executable (.exe) file [Cameyo 2012]. Unlike CDE, which monitors execution, Cameyo monitors the installation process. It offers two mechanisms for accomplishing the virtualization. The first is to take a snapshot of the system, then install the application, take another snapshot, and compute the dependencies and modified registry keys from the difference between these snapshots (similar to the VM overlay creation approach for VM synthesis). The second mechanism does not take snapshots but instead simulates the installation process, keeping track of all of the installer’s actions. This simulated installation does not have any permanent effect on the actual system.

If the application relies on anything that the installer does not provide, it must be added manually to the package. A Cameyo package includes its own directory structure and registry. The runtime environment within the package redirects file and registry accesses into the package. This sandbox can be configured to permit access to files outside the package.

6 Implementation

This section describes the implementation of the application-virtualization-based cloudlet cyber-foraging system.

6.1 Basic Architecture

The two main components of the architecture are the mobile device and the cloudlet host as a machine that lends its resources to the mobile device, as shown in Figure 7. All devices are connected to the same subnet within a wireless network. The cloudlet host runs a hypervisor to host multiple VMs. The different VMs provide a selection of various operating systems and versions. The mobile device may select one of these VMs as a target system for its application offload. Each VM runs a *cloudlet server* that publishes information to the network about the VM's operating system and other relevant properties. The *cloudlet client* that runs on the mobile device collects this information. Every cyber-foraging-enabled application is divided into a client and a server. The client is designed to run on the mobile device and the server is to be offloaded to the cloudlet. When the user decides to start a cyber-foraging-enabled application, and the cloudlet client can find a suitable cloudlet system, the cloudlet client transmits the server part to the selected cloudlet server. The server part consists of two items: the *application metadata* and the *application package*. The application metadata is the information that is relevant for the offload operation; the application package is a compressed archive that contains the executable server along with any necessary data. After receiving the application package from the mobile cloudlet client, the cloudlet server prepares the retrieved application server for execution. After signaling the successful start of the application server to the mobile device, the cloudlet client starts the application client, which then can interact with the application server.

6.1.1 Mobile Device

The mobile device is runs Android 4.1 and supports multicast, which is required for the discovery mechanism. All the elements of each cyber-foraging-enabled application are stored on the mobile device: the application client, the application metadata, and the application package that contains the application server.

6.1.2 Cloudlet Host

The cloudlet host is a multicast supporting machine that runs the VM hypervisor.

6.1.3 VM Hypervisor

We use KVM, which is a common and mature hypervisor for VMs that is part of the Linux kernel. The KVM-managed VMs connect to the network in bridged network mode, which means that a VM has its own IP address [KVM 2011].

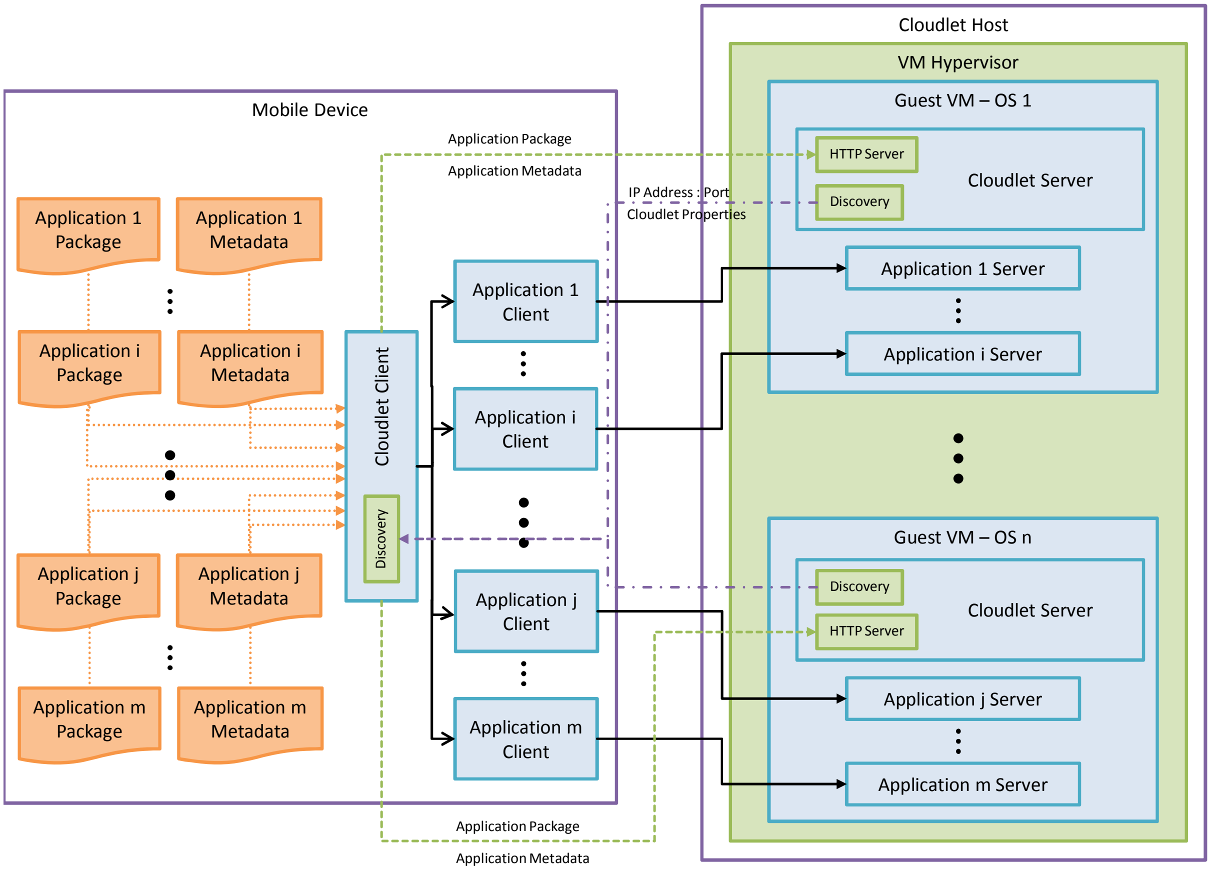

Figure 7: Application-Virtualization-Based Cloudlet Cyber-Foraging System Architecture

6.1.4 Cloudlet Client

The cloudlet client is an Android 4.1 application. It searches the mobile device's storage at a dedicated location for cyber-foraging-enabled applications and displays them in a list. It is also responsible for discovering cloudlet servers. When the user selects to run one of the applications from the list, the cloudlet client transmits the application metadata and application package via HTTP to an appropriate cloudlet server. We will discuss the process of finding an appropriate cloudlet in Section 6.3.2. The cloudlet client application displays upload progress and shows status information that is retrieved from the cloudlet server. After successful deployment on the cloudlet, the cloudlet client starts the application client.

6.1.5 Cloudlet Server

The cloudlet server is a Java program that requires Java SE Runtime Environment 7 (JRE 7) or higher. It contains a Jetty HTTP server (2012) that is responsible for processing file uploads and sending status messages to the cloudlet client. It registers its service as a cloudlet server by sending its service information via multicast. Although the cloudlet server can run on any operating system that supports Java, it does rely on OS-specific code for package decompression and terminal execution. When the cloudlet server is started, it detects the underlying operating system automatically and chooses the code to use for these tasks.

6.1.6 Discovery

The discovery mechanism is provided by the JmDNS library [van Hoff 2011], which is a pure Java implementation of multicast Domain Name Systems (DNS) and the zero configuration networking (zeroconf) framework [zeroconf 2012]. The cloudlet server registers a service, thereby publishing information about itself. The cloudlet client uses JmDNS for exploring services that are published in the zeroconf multicast group and adds newly discovered services to its internal list of cloudlet servers.

6.1.7 Application Client

Each cyber-foraging-enabled application consists of a client and a server. The application client is an Android application. After successful deployment of the server on the cloudlet, the cloudlet client launches the application client's main activity. In doing so, it submits the address and port number of the application server as parameters. The application client can then connect to the application server on the cloudlet in order to submit tasks and receive results.

6.1.8 Application Server

The application server is the executable that is the counterpart of the application client. It receives tasks from the application client that the application server then carries out on its behalf. Afterwards, the computational result is sent back to the client. For example, the face recognition server in the cloudlet scenario that Section 3.3 describes receives images from the corresponding mobile application client and responds with a list of recognized faces.

6.1.9 Application Package

The application package is a compressed archive that contains the application server and all dependencies that are necessary for deployment on a cloudlet. A concrete example of an application package is a gzipped tarball that contains a CDE package. The CDE package holds the server executable, libraries, environment settings, and other necessary files.

6.1.10 Application Metadata

A Javascript Object Notation (JSON) file that contains information about the package accompanies every application package [Crockford 2006]. The application metadata includes all cloudlet requirements that must be met for a successful offload (see Section 6.3.2).

6.2 Application Deployment Sequence

Figure 8 is a Unified Modeling Language (UML) sequence diagram that shows the interaction between the mobile device and the cloudlet that takes place during application deployment. The participating actors are the cloudlet client with its JmDNS-based discovery component and the cloudlet server. After successful deployment, the application client (":Activity") starts its interaction with the application server (":Process"). All mobile to cloudlet interaction is done via HTTP requests and responses. The protocol between the application client and the application server is implementation specific.

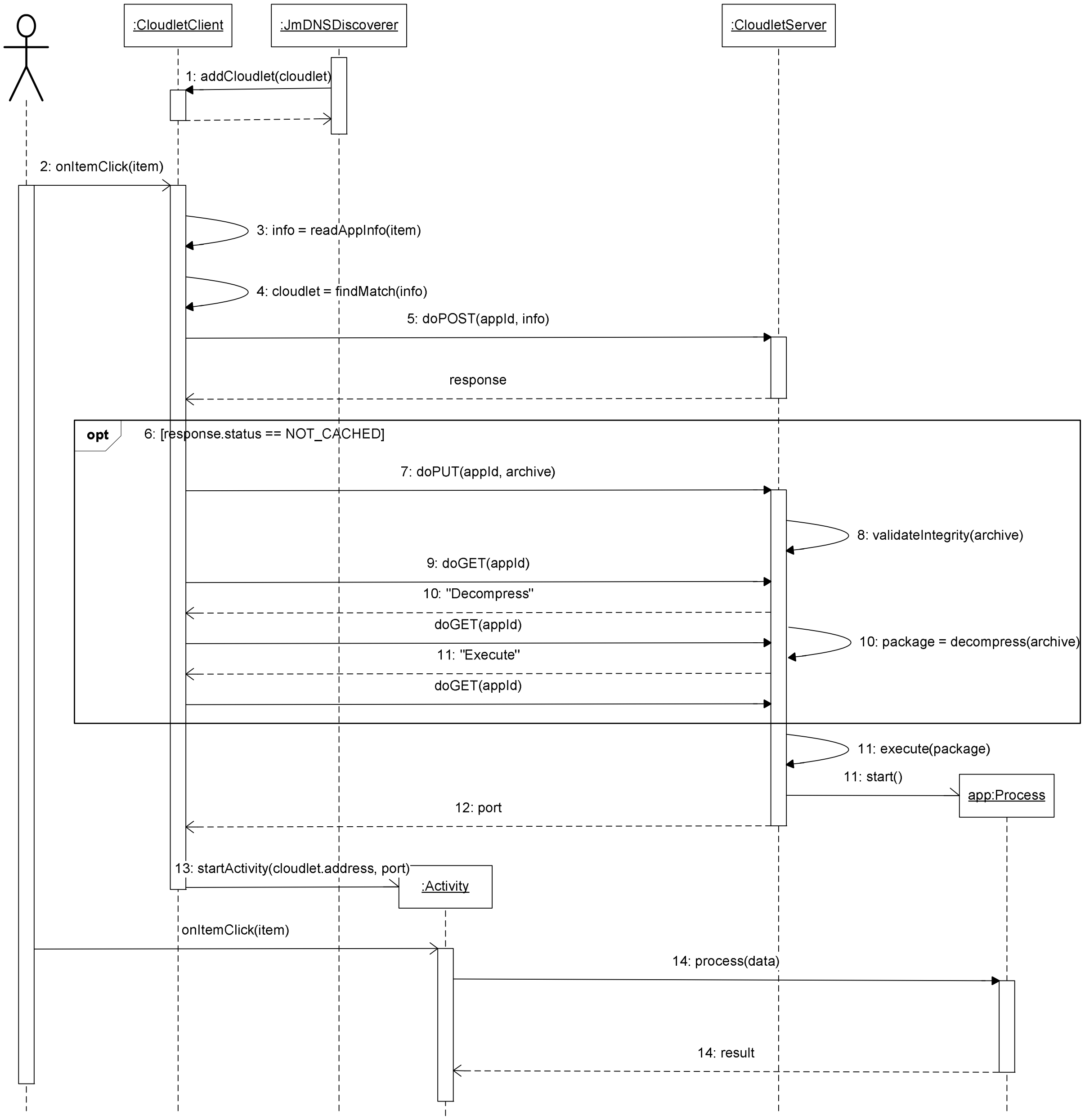

Figure 8: Application Deployment on a Cloudlet

1. The JmDNSDiscoverer finds a new Cloudlet service and adds the information to the CloudletClient’s list of available cloudlet servers.
2. The user clicks on one of the cyber-foraging-enabled applications in the list.
3. The application metadata is extracted from the application.
4. A cloudlet that matches the application requirements that are contained in the application metadata is selected from the list of available cloudlet servers.
5. The cloudlet client issues an HTTP Post request to the selected CloudletServer. This request contains a unique application identifier and the application metadata.

6. If an application with this identifier is already deployed on the cloudlet, go to Step 12.
7. Because the application is not cached on the cloudlet, the CloudletClient transmits the application package (“archive”) to the CloudletServer.
8. On the cloudlet, the Cloudlet Server validates the integrity of the application package by comparing the md5 checksum and file size to the values contained in the application metadata.
9. The CloudletClient communicates with the CloudletServer to receive upload progress and status information by sending GET messages. Every time it receives a response, it immediately sends a new request (GET message) so that the CloudletServer can push messages to the CloudletClient after every step of the deployment process.
10. The CloudletServer decompresses the application package archive and informs the CloudletClient about its activity.
11. The CloudletServer sends the “Execute” progress status to the CloudletClient and starts a new system process for the application server.
12. The application is ready and the CloudletServer sends the port on which the application server operates to the CloudletClient.
13. The CloudletClient starts the application client with the given port and cloudlet address.
14. The application client (“:Activity”) sends data to the application server (“app:Process”). The application server processes this data and returns the result to the application client on the mobile device.

6.3 Implementation Details

This goal of this section is to provide deeper insight into the implementation performed in the context of this technical note. To this end, we present and discuss selected design decisions.

6.3.1 Cloudlet Server Code View

The cloudlet server code turns a VM into a code offload site. The code is completely written in Java 7 to ensure execution on different operating systems. The cloudlet server code is divided into different packages with each one fulfilling a single part of the cloudlet server’s tasks (see Figure 9). The *server* package contains classes that process requests from the cloudlet client. This functionality is provided by embedding a basic Jetty HTTP server that maps HTTP requests to corresponding HTTP servlets that process client requests. The *jmdns* package enables service discovery by registering the cloudlet service in the network using JmDNS, which is a Java implementation of the zeroconf networking technique. The instructions regarding how to process uploaded applications are bundled inside the *packagehandler* package. This package contains interfaces and abstract classes that must be implemented for each application file type and required operating system. For this technical note, Windows and Linux packagehandlers have been implemented. The *server* and *packagehandler* packages both use the *fileprocessing* package, which can copy and delete files, compute checksums, and decompress archive files.

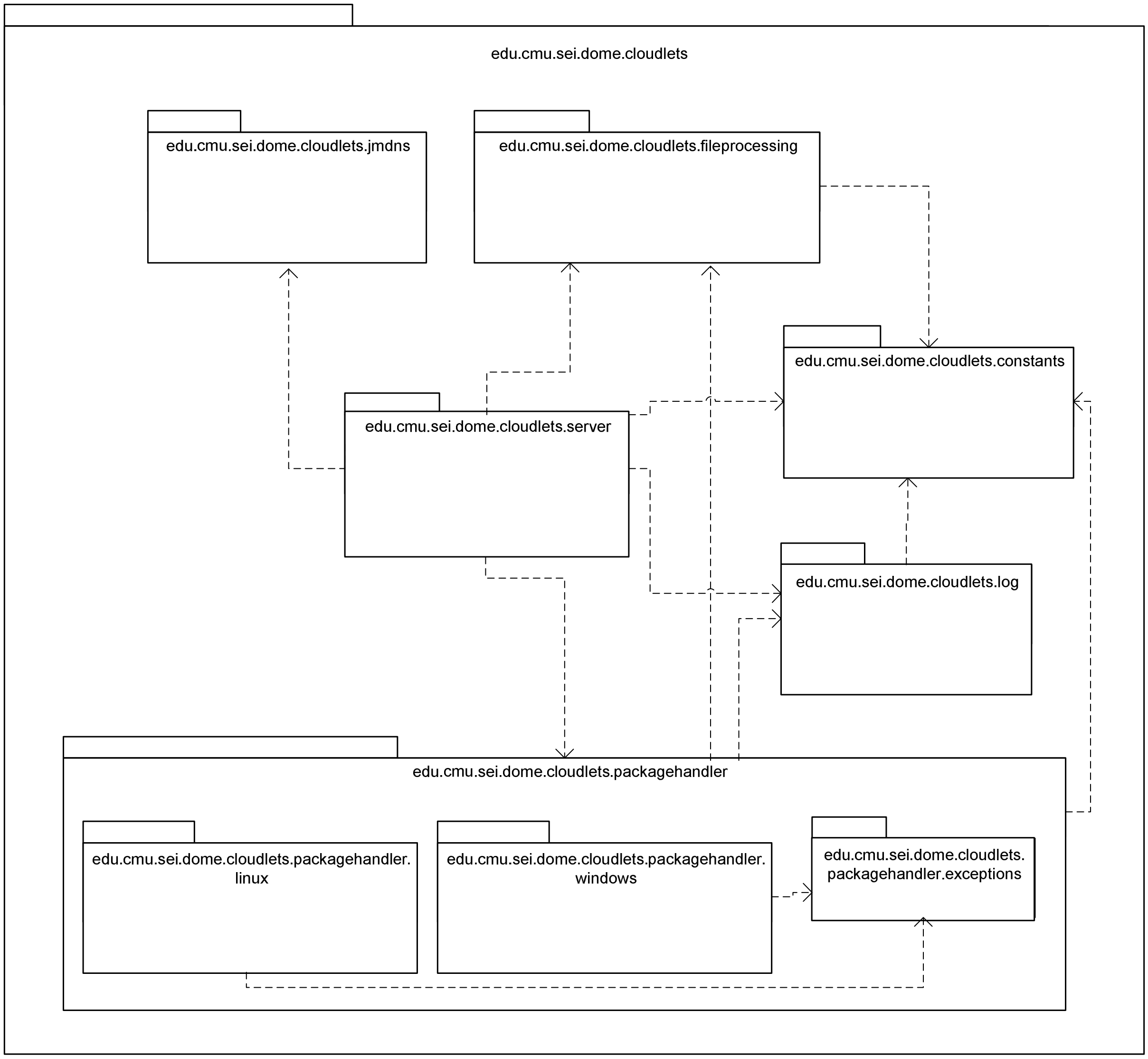

Figure 9: Cloudlet Server Package View

```
{
    "name": "application metadata",
    "properties": {
        "name": {
            "type": "string",
            "description": "Name of the application",
            "required": true
        },
        "description": {
            "type": "string",
        },
        "checksum": {
            "type": "string",
            "description": "md5 hash of the application package",
            "required": true
        },
        "size": {
            "type": "number",
            "description": "File size of the application package in bytes",
            "required": true
        },
        "type": {
            "type": "string",
            "description": "application package type, e.g. cde",
            "required": true
        },
        "package": {
            "type": "string",
            "description": "package name of the Android application client",
            "required": true
        },
        "port": {
            "type": "number",
            "description": "Port on which the application server listens",
            "required": true
        },
        "server_args": {
            "type": "string",
            "description": "Command line arguments for the application server"
        },
        "cloudlet": {
            "type": "array",
            “description": "Set of cloudlet properties. The properties             may
                  be arbitrary key-value pairs or minimum or maximum numeral
                  requirements, e.g. cores_min: 4",
            "items": {
                "type": "object"
             }
         }
    }
}
```

Figure 10: JSON Schema for the Application Metadata File

6.3.2 Application Metadata and Cloudlet Requirements Matching

The application metadata that accompanies each application package provides the information necessary to execute the cyber-foraging process. It is a JSON file, and the schema in Figure 10 describes its structure. The *name* and *description* fields hold general information about the application; the cloudlet server uses *checksum* and *size* to guarantee the binary integrity of the uploaded application package. The checksum value is the md5 hash of the application package archive. *Type*, *port* and *server_args* inform the cloudlet server on how to handle the application package. Possible types that we used for our implementation are “cde,” “cameyo,” “jar,” and “exe.” *Server_args* is an optional field; if it is set, the *server_args* string value will be split into command line arguments that will be used to start the application server. The *package* field defines the Android application client; its value enables the cloudlet client to start the corresponding Android activity.

While all other fields have basic types, *cloudlet* is an array of JSON objects. Each of these objects defines the set of requirements that must be satisfied by a cloudlet in order to execute the application. If and only if a cloudlet’s properties match one of these sets, it is eligible to serve the mobile device. The cloudlet properties are defined in its own JSON file, which is read and evaluated by the cloudlet server. Cloudlet properties are published via the JmDNS service registration, thus enabling the cloudlet client to find a matching cloudlet server. Cloudlet properties match a set of requirements if and only if

- every field in the set can be found in the cloudlet properties with the same value, except for
  - a number field `<name>_min` must be met by a cloudlet field `<name>` with a number greater or equal
  - a number field `<name>_max` must be met by a cloudlet field `<name>` with a number lower or equal

The example in Figure 11 illustrates this principle. In this example, the properties of Cloudlet 1 (Windows 7 OS, x86-64 architecture and 8 cores) match the second set of cloudlet requirements (Windows 7 OS, x-86-64 architecture, and a minimum of 4 cores), which means that the cloudlet is a valid offload site for the application server.

```
"cloudlet": [
    {
        "os": "windows xp",
        "architecture": "x86"
    },
    {
        "os": "windows 7",
        "architecture": "x86-64",
        "cores_min": 4
    }
]
```

Sets of cloudlet requirements

```
{
    "os": "windows 7",
    "architecture": "x86-64",
    "cores": 8
}
```

Properties Cloudlet 1

```
{
    "os": "fedora 16",
    "architecture": "x86-64",
    "cores": 4
}
```

Properties Cloudlet 2

Figure 11: Cloudlet Requirements Matching

6.3.3 RESTful Architecture

Representational State Transfer, also known as *REST* , is a Web service design architecture that is centered on the concept of *resources* [Fielding 2000]. A client can access and modify these resources through a uniform interface provided by the server. This interface uses the HTTP methods GET, PUT, POST and DELETE.

The cloudlet implementation takes advantage of the REST principles to provide an easily understandable pattern for application management on the cloudlet. In this case, the resources are the application servers that are addressed via the checksum of the application package. Given the cloudlet server's address and port, the address for a resource is http://<address>:<port>/apps/<checksum>. This address scheme should provide each resource with a unique identifier.[1] Table 1 shows the effect of each HTTP request on a resource in the context of the cloudlet server implementation.

Table 1: RESTful Service Interface for Application Management on the Cloudlet

GET	POST	PUT	DELETE
Get a message with the current status of the application.	Transmit the application metadata JSON file to create an entry for the application on the cloudlet.	Store, decompress, and execute the transmitted application package. Depending on the cloudlet configuration, replace any representation of the same application if exists.	Delete the representation and entry of the application.

6.3.4 Long Polling

Using HTTP between a client and a server enforces a strict request-response protocol in which the client initiates all communication. There is no persistent connection between client and server; once the request has been answered (i.e., the server sends the corresponding response to the request), the HTTP connection terminates. As a result, the server can only communicate with the client in the context of a response to a client request. However, in some cases it is useful to let the server push messages to the client, instead of waiting for the client to pull information via HTTP requests.

One technique to allow the server to send messages to a client is *long polling*. In our cloudlet implementation, the server sends messages to the client about application deployment progress using this technique. Long polling emulates a server push mechanism: an HTTP request is not served immediately but is "held back" by the server. When the server must send a message to the client, it responds to this held-back request. As soon as the client receives the response, it immediately initiates a new request that is then held back again by the server until it decides to respond.

The cloudlet client starts to activate long polling immediately after transmitting the application package to the cloudlet server. The cloudlet server sends progress status notifications to the cloudlet client, indicating the server action to be performed next. The stages on the server side are validation (integrity check), decompression, and execution of the offloaded application. The user of the mobile device can therefore track the status of application deployment, which otherwise would not be possible.

---

[1] There is research that shows that there may be collisions between md5 hashes (How to Break MD5 and Other Hash Functions, 2005), which is why they cannot guarantee uniqueness. A more advanced cryptographic hash function could minimize this risk.

The Jetty HTTP framework supports long polling through a concept called *Continuations*. A continuation encapsulates an HTTP request and suspends it. When the corresponding response is sent, the continuation is considered completed.

In our implementation the long polling client is the *EventListener* class, which is a subclass of *Thread*. The status code of an HTTP response determines how the response is handled. A status that is neither 400 (“ERROR” status in Figure 12) nor 410 (“FINISH” status) causes the EventListener to issue a new GET request. Figure 12 shows an excerpt of the *EventListener* source code.

```
@Override
public void run() {
    while (running) {
        try {
            HttpGet get = new HttpGet(url);
            HttpResponse response = client.execute(get);
            String content = HttpUtil.getContent(response);
            if ((response != null) && (response.getEntity() != null))
                showResponse(content);

            if (response.getStatusLine().getStatusCode() == ERROR) {
                stopListening();
                // error handling
                // ...
            }
            // no follow up - server finished 'connection'
            else if (response.getStatusLine().getStatusCode() == FINISH) {
                stopListening();
                // retrieve port and start application client
                // ...
            }
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            cloudletClient.error("Could not reach " + url + "!");
        }
    }
}
```

Figure 12: Listing 1: Client Long Polling - EventListener.java

The server counterparts are the *RESTservlet* and *PushHandler* classes. The *RESTservlet* encapsulates a request into a *Continuation* and suspends it. This *Continuation* is then passed to the *PushHandler* that completes it when a message should be pushed to the client. The main functionality of these classes is presented in Figure 13 and Figure 14.

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    // get application ID, strip first character, i.e. slash
    String appId = req.getPathInfo().substring(1);
    Continuation continuation = (Continuation) ContinuationSupport
            .getContinuation(req);
    continuation.suspend(resp);
    PushHandler push = PushHandlerStore.getPushHandler(appId);
    push.addRequest(continuation);
}
```

Figure 13: Listing 2:Server Long Polling: RESTservlet.java

```
private void pushToClient(String message, int status) throws
         IOException {
    if (message == null || message.equals(""))
        return;
    Continuation continuation = waitForClientRequest();
    if (continuation == null)
        return;

    Log.println(appId, "Respond: " + message);
    HttpServletResponse resp = (HttpServletResponse) continuation
            .getServletResponse();
    resp.setContentType("text/html");
    resp.setStatus(status);
    resp.getWriter().write(message);
    continuation.complete();
}
```

Figure 14: Listing 3: Server Long Polling - PushHandler.java

6.3.5 Bridge Pattern for OS Decoupling

A goal of the cloudlet server design is to be portable across a variety of operating systems. While the HTTP server and JmDNS service functionality is independent of the OS and the application, the handling of application packages relies on OS- and application-type-specific behavior. In order to support extensibility to more cloudlet environments, a good practice is to separate the OS- and application-specific code from the portable part of the program.

In our implementation, we achieve this through the Bridge design pattern [Gamma 1995]. The Bridge pattern decouples abstraction from implementation, thus facilitating changes to the implementation without having to change the code that binds to the abstraction.

The *PackageHandler* class serves as the abstraction part of the pattern. It has an instance of an implementation of the *PackageHandlerImpl* interface, which encapsulates all OS- and application-specific code. Calls to the *PackageHandler decompress* and *execute* methods are delegated to the concrete *PackageHandlerImpl*. The Bridge pattern allows for an abstraction hierarchy that is independent from the hierarchy on the implementation side and provides extension points for both.

Each operating system family that is to be supported by the cloudlet server must implement the *PackageHandlerImpl* interface and provide the OS- and application-specific code. The implemented cloudlet server includes the *LinuxPackageHandler* and the *WindowsPackageHandler for Linux- and Windows-based applications*, respectively. The application-specific classes inherit from the abstract *Executor* class that is responsible for starting an application with provided arguments. For example, the *LinuxPackageHandler* uses the *CDEExecutor* and *JARExecutor* classes, which encapsulate knowledge on how to handle CDE or JAR (Java Archive) packages, respectively. Both classes extend the *LinuxTerminalExecutor* that starts a terminal process that runs the CDE or JAR application. Figure 15 shows the entire *PackageHandler* Bridge implementation as a UML class diagram.

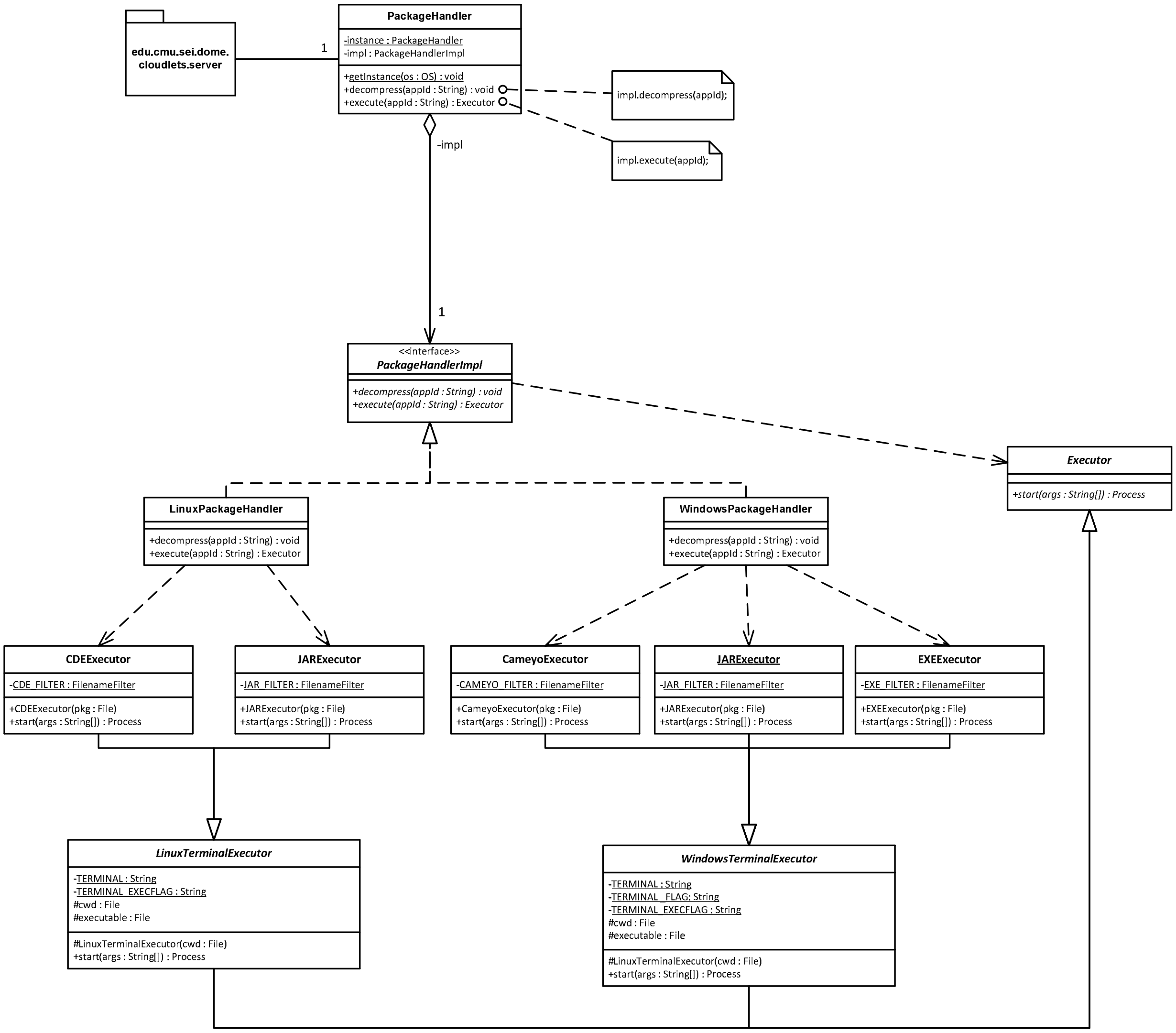

Figure 15: The Bridge Pattern Decouples the PackageHandler Abstraction from OS-Specific Implementations

7 Evaluation and Comparison of Application Virtualization and VM Synthesis

7.1 Functional Requirements

Based on the functional requirements presented in Section 3.5.1, both application virtualization and VM synthesis are valid strategies for cyber foraging. They both meet requirements for cloudlet use in hostile environments:

- They are based on stateless servers that do not rely on internet access for provisioning—both cloudlet implementations receive the application from the mobile client.
- Preceding deployment phase is a cloudlet discovery phase, in which the mobile device finds suitable cloudlets by parsing the service information published by the cloudlets.
- Correct execution of offloaded application is guaranteed as long as the package or overlay has been created correctly.
- They are able to return to a state with no traces of an offloaded application, that is, a complete removal of the application.
- They can serve multiple clients simultaneously.

7.2 Quantitative Analysis

To understand battery efficiency and performance, we evaluated the application-virtualization-based cloudlet implementation using the following applications.

Object Recognition

The application server is a Linux C++ program that receives a camera input image from the Android application client and returns a list of objects that could be recognized in the image. The object recognition is based on MOPED [CMU 2011]. CDE was used to virtualize the application server.

Speech Recognition

Based on SPHINX [CMU 2012], the speech recognition server is a Java program that was virtualized for Linux environments with CDE and for Windows with Cameyo. It receives a WAV file from the Android application client and returns the recognized input as text.

Face Recognition

Based on OpenCV [Itseez 2012], the Face Recognition application server is a C++ program for Windows and was virtualized with Cameyo. It continuously receives camera input from the Android application client and returns the areas in which it could find a face where there is a match from the internal database.

NULL

Virtualized with CDE for Linux and with Cameyo for Windows, the NULL application server is a C program that returns immediately after start. There is no Android application client. The NULL application is used to determine the baseline for transmission overhead and battery consumption.

Table 2 shows both the original application size and the size of the compressed virtualized application.

Table 2: File Sizes of Applications and Compressed Application Packages

	Application Size (MB)	**Compressed Package Size (MB)**	
		CDE	Cameyo
Object Recognition	25.340	28.492	-
Speech Recognition	100.140	67.748	65.370
Face Recognition	34.449	-	13.090
NULL	0.009	1.133	0.940

7.2.1 Experiments

We conducted the experiments using a Galaxy Nexus mobile device running Android 4.1.1 and an 8 core, 2.00 GHz Intel Xeon, 32 GB RAM machine that served as a cloudlet host (see Figure 16). The wireless network was an 802.11n Wi-Fi network at the frequency of 5 GHz. The cloudlet machine hosted two VMs: Ubuntu 10.04 and Windows XP.

For measuring the mobile device's energy consumption, we used the Power Tool device and corresponding software from Monsoon Solutions [Monsoon 2008].

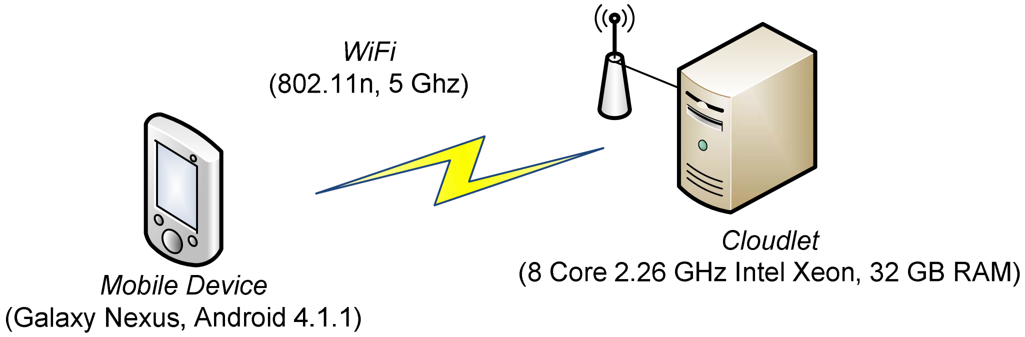

Figure 16: Evaluation Experimental Setup

Table 3 and Figure 17 show the average time measurements for each deployment process step and the total energy consumption per application.

Table 3: Time Measurements (s) and Energy Consumption (J) per Virtual Application

	Metadata Transmission (s)	Application Transmission (s)	Save to Disk (s)	Validation (s)	Decompression (s)	Application Start (s)	Energy (J)
Object (CDE)	0.197	15.445	0.091	0.191	1.351	0.210	38.484
Speech (CDE)	0.113	24.329	0.324	0.482	1.868	0.212	56.075
NULL (CDE)	0.100	0.576	0.004	0.008	0.064	0.209	1.958
Face (Cameyo)	0.250	6.695	0.659	2.918	1.089	5.127	33.641
Speech (Cameyo)	0.113	41.219	2.228	1.0126	4.941	16.656	98.118
NULL (Cameyo)	0.100	6.706	0.003	0.009	0.081	2.310	14.940

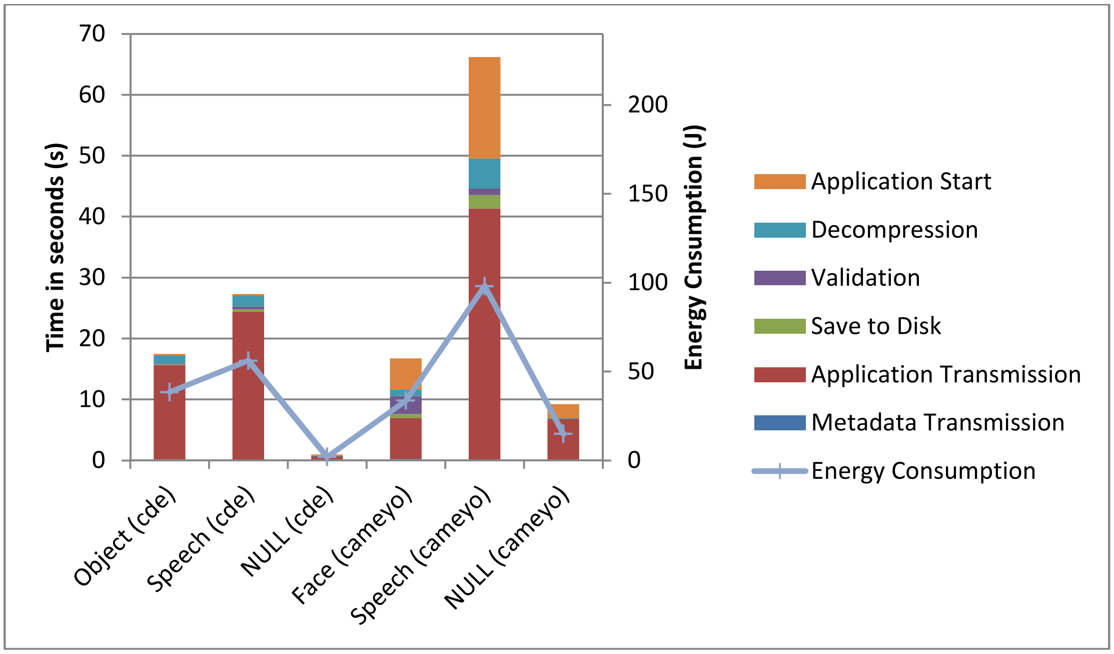

Figure 17: Time and Energy Measurements per Virtual Application

Deployment time ranged from 0.961 seconds (*NULL* (*CDE*)) to 66.170 seconds (*Speech* (*Cameyo*)). *NULL (CDE)* had the lowest energy consumption, which was approximately 2 Joules. *Speech* (*Cameyo*) consumed the highest amount of energy at approximately 98 Joules. Application transmission time clearly dominates the total deployment time for CDE applications.

It is also the major portion for Cameyo applications. Application start time is significant for Cameyo applications and negligible for CDE applications.

7.2.2 Conclusions

We divided the applications into two groups because of the observed difference in performance. CDE applications that run on an Ubuntu 10.04 VM form the first group and Cameyo applications for Windows XP form the second group.

There is a strong positive correlation between application package size and deployment time for both CDE/Ubuntu10.04 and Cameyo/WinXP applications. The correlation coefficients are

$$r_{size,time}(CDE, Ubuntu10.04) = 0.96940301$$

$$r_{size,time}(Cameyo, WinXP) = 0.99840802.$$

These values therefore suggest a linear dependency between package size and deployment time. An explanation for this observation is the linear dependency that also exists between file size and file transmission time, which mostly is determined by the Wi-Fi bandwidth. In addition, there is a proportional relation between file size and the time needed for checksum computation and decompression.

The measurements also suggest that Cameyo's application start time is proportional to file size (see Figure 18 and Figure 19). The linear regression equations for this relationship are

CDE/Ubuntu 10.04: $time_s = 0.3587 \times file\ size_{MB} + 2.7378$

Cameyo/WinXP: $time_s = 0.9204 \times file\ size_{MB} + 6.8220$

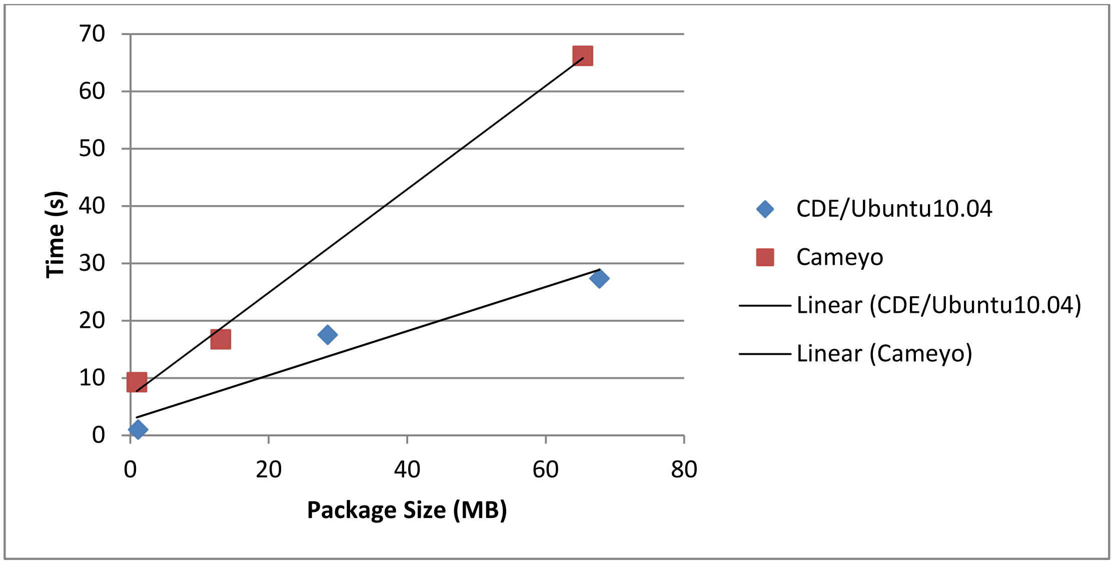

Figure 18: Application Package Size in Relation to Deployment Time

The experiments showed the highest energy consumption during application transmission. Given the linear relation between application transmission time and package size, there is also a linear dependency between package size and energy consumption, as shown in Figure 19. The correlation coefficients are

$$r_{size,energy}(CDE, Ubuntu10.04) = 0.95472387$$

$$r_{size,energy}(Cameyo, WinXP) = 0.99929470$$

and the linear regression equations for this relationship are

$$\text{CDE/Ubuntu 10.04:} \qquad energy_J = 0.7871 \times file\ size_{MB} + 6.6252$$

$$\text{Cameyo/WinXP:} \qquad energy_J = 1.2738 \times file\ size_{MB} + 15.1870$$

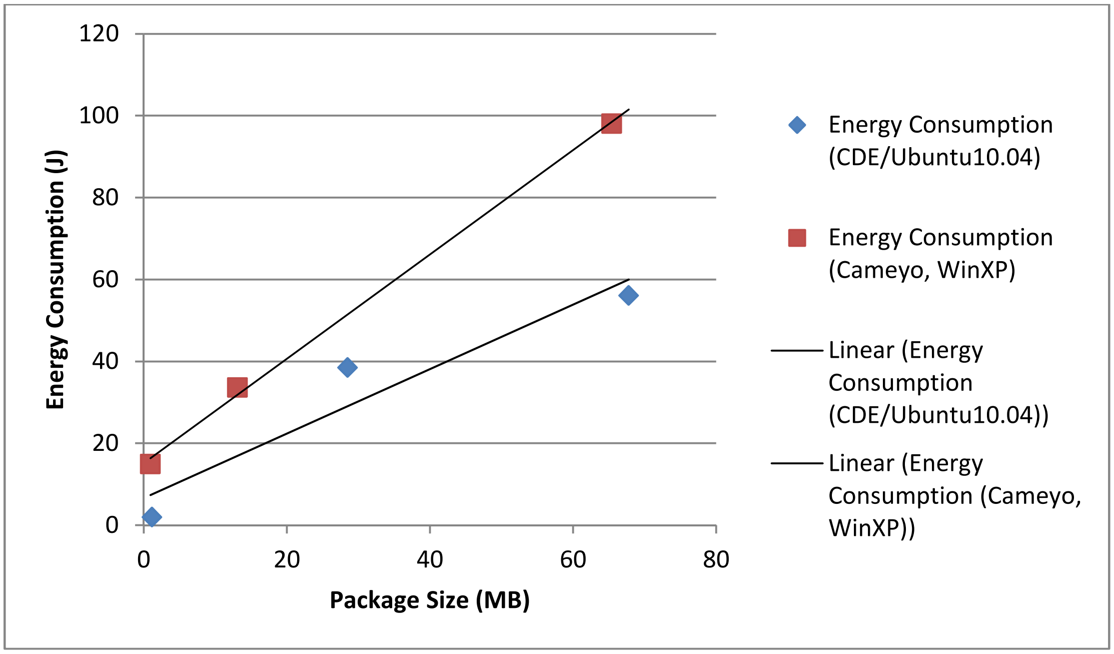

Figure 19: Application Package Size in Relation to Energy Consumption

7.2.3 Comparison with VM Synthesis

Simanta and colleagues have evaluated the reference architecture of the VM synthesis cloudlet, using the same set of applications [Simanta 2012]. As in the application virtualization cloudlet implementation, energy consumption and deployment time (referred to as “application ready time” by Simanta and colleagues), increases with the amount of data that is transferred to the cloudlet. Based on the numbers from the revised VM synthesis prototype evaluation [Simanta 2012, p. 18], a linear regression analysis shows a relation of

$$energy_J \approx 0.82 \times file\ size_{MB} + 7.93$$

$$time_J \approx 0.68 \times file\ size_{MB} + 6.31.$$

where “$file\ size_{MB}$” is the total amount of data that is transmitted, that is, the sum of disk image overlay size and memory overlay size.

The experiments and the VM synthesis evaluation clearly show that smaller packages for transfer can drastically decrease deployment time and energy consumption for cloudlet-based cyber foraging. Using the software from the VM synthesis implementation from a report by Simanta and colleagues, we created disk and memory overlays of the same applications that were used for evaluating the application virtualization implementation [Simanta 2012]. In this set of experiments, we used 64-bit versions of Ubuntu 12.04 and Windows XP with Service Pack 2. The results show that the virtualized applications are significantly smaller than the combined overlays (see Table 4). The main reason is that memory overlays tend to include data that is irrelevant to the application of interest, thus leading to an increased file size.

Table 4: File Sizes of Application Virtualization Versus VM Synthesis

	NULL		**Object**	**Speech**		**Face**
	Linux	Windows	Linux	Linux	Windows	Windows
Compressed virtualized application	1.1 MB	0.9 MB	28.5 MB	67.7 MB	65.4 MB	13.1 MB
Compressed disk overlay	0.1 MB	0.4 MB	42.8 MB	104.8 MB	113.7 MB	33.5 MB
Compressed disk + memory overlay	21.2 MB	4.2 MB	144.5 MB	226.8 MB	425.7 MB	141.5 MB

Application package size dominates application ready time and energy consumption for both VM synthesis and application virtualization. The factors in the relation of time and energy to file size for VM synthesis are comparable to those for application virtualization. Consequently, because file size of a virtual application clearly is smaller than file size of the overlays for the same application, application virtualization outperforms VM synthesis in terms of fast deployment and low energy consumption.

7.3 Qualitative Analysis

This section discusses qualitative aspects of both solutions and emphasizes the tradeoffs between strategies. Section 7.4.7 provides a summary of the analysis.

7.4 Coupling Between Application and Cloudlet

For the provision of a general cloudlet infrastructure that is capable of serving a large variety of applications, the coupling between application and cloudlet is a significant factor. Loose coupling enables the setup of cloudlet hosts that can act as surrogates to a large number of offload-ready applications. Tight coupling, on the other hand, requires cloudlet hosts to provide more specialized environments that fit the particular application.

The goal of *application virtualization* is to separate applications from the underlying operating system. The implementation achieves this goal with respect to portability across distribution boundaries; for example, a CDE application runs on various Linux distributions without having to adapt the virtualized application. However, applications cannot cross operating system family boundaries; for example, CDE does not run on Windows and Cameyo does not run on Linux. This limitation occurs because a virtualization runtime is bound to a special underlying set of system calls.

VM synthesis requires a target system on the cloudlet that is the binary equal of the source system on which the application was made ready for offload. The part of the system that is transferred to the cloudlet is the binary difference between two VM snapshots. Hence, in order to restore the final VM image, the cloudlet host must store the first snapshot, that is, the base VM image. As a result, there is tight coupling between application and cloudlet when using VM synthesis because the mobile device requires the cloudlet host to have the correct base image. A potential workaround is to have the mobile device transmit the complete final VM image, but this would lead to high costs regarding memory storage, deployment time, and battery consumption because of the large size of the image. However, this option would offer much looser coupling than application virtualization because the cloudlet would only need to run a hypervisor that can handle the transferred images.

The bottom line is that both solutions are coupled to some aspect of the system. Application virtualization is coupled to the operating systems and distributions supported by the cloudlet. VM synthesis offers greater flexibility in terms of operating systems and distributions, but is coupled to the base VM that was used for overlay creation and has a higher cost in terms of transmission times and energy consumption due to larger transfer sizes.

7.4.1 “Patchability” of the Target System

In VM synthesis, base VMs cannot be updated without invalidating corresponding overlays (i.e., overlays created using the base VM). To provide a secure and stable system, regular system updates (or patches) are necessary. However, every system update requires a recreation of overlays and a distribution of these overlays to mobile devices. The other option is to provision cloudlets with multiple sets of base VMs in order to keep supporting legacy overlays. This contradicts the original idea of an easy-to-deploy cloudlet that is general enough to host many applications.

Application virtualization enables the cloudlet host to provide operating systems that can be updated without affecting the execution of virtualized applications. This remains true as long as the updates do not conflict with the application virtualization runtime environment itself. The application virtualization runtime environment prevents such conflicts by relying only on very basic OS functionality.

Application virtualization therefore supports system patches better than VM synthesis does.

7.4.2 Range of Offload-Ready Applications

As the authors of CDE explain, applications “that require specialized hardware or device drivers” [Guo 2011, p. 7] cannot be made portable across machines. Cameyo can package device drivers and temporarily integrate them into the operating system [Cameyo 2012]. However, this approach only works for drivers that do not address the application’s files and registry that are hidden within a sandbox. The device driver itself is not virtualized and runs outside the sandbox.

Unlike application virtualization, VM synthesis does not have any issues with device drivers because the VM overlay includes all drivers that have been added to the base VM. However, applications that use VM synthesis as its distribution mechanism would expect the cloudlet to have specialized hardware if it were required.

The result is that the range of offload-ready applications for VM synthesis is broader than the range for application virtualization because application virtualization does not support hardware-related functionality.

7.4.3 Correct Operation

It is important to guarantee the correct operation of tasks that are offloaded to a cloudlet. Such is especially important in hostile environments where the reliability of tools is often essential to a mission's success.

VM synthesis simply mirrors the application's original functionality by reconstructing the entire operating system under which the application has been installed. If this installation has been faultless, the offloaded application operates correctly as well. As mentioned in the previous section, the cloudlet may have to provide special hardware requirements. These requirements must be documented and then must be negotiated with potential cloudlets.

Virtualized applications behave correctly as long as all of their dependencies can be met by the execution environment. This implies that all dependencies that should be portable are included in the application package. It is also possible to virtualize only parts of an application and take advantage of other components that are installed on the cloudlet, for example, runtime environments such as the JRE or device drivers. If one of these components appears to be incompatible, however, the execution will fail.

VM synthesis guarantees correct operation because it creates a replica of the original system. Application virtualization involves the risk of missing dependencies, which may cause malfunctions when the application is ported to another system. However, if all dependencies are packaged, correct operation can be guaranteed.

7.4.4 Application Preparation Overhead

Preparing an application for deployment on a cloudlet should require only a small amount of effort. VM synthesis and application virtualization do not require source code modifications or insight into the application's source code. This means that application developers need not be involved in offload preparation.

VM synthesis requires the installation of the application on a base VM and then computes the overlay as a binary difference between the suspended state of the complete VM and the base VM (see Figure 20). This is a convenient mechanism, where the main difficulty would be application installation.

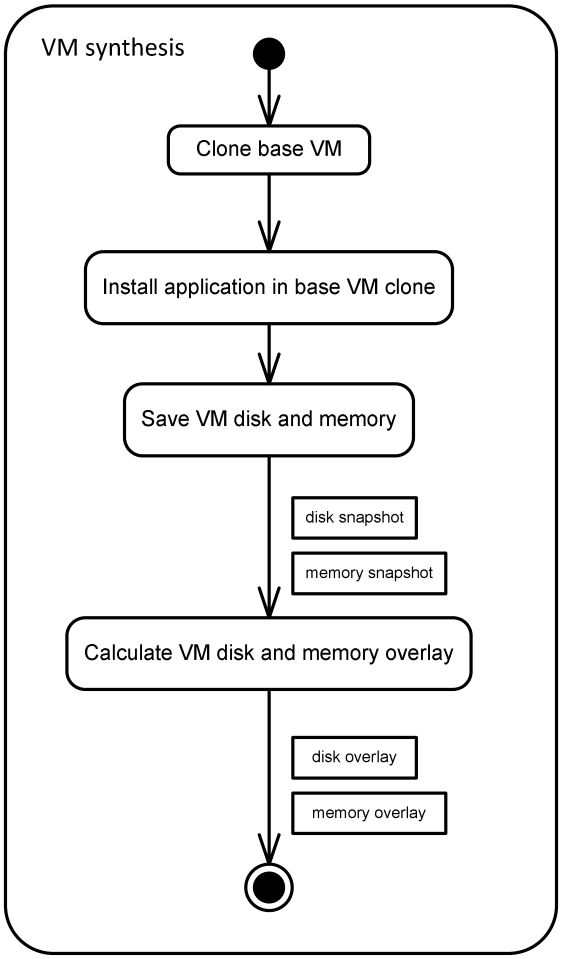

Figure 20: VM Overlay Creation Process for VM Synthesis

Application virtualization is achievable in various ways, as shown in Figure 21. CDE copies the current environment settings and supervises the application's execution during runtime in order to package all files that have been involved in the execution. Cameyo supervises the installation process instead. Either it compares system snapshots from before and after the installation, or it emulates the installation routine itself. Both tools allow the creation of packages from scratch or the modification of a created package to add missing files or dependencies. This is often necessary because the original supervision routines cannot guarantee finding all dependencies (see Section 5.2). Therefore, application virtualization requires deeper knowledge of the application's dependencies.

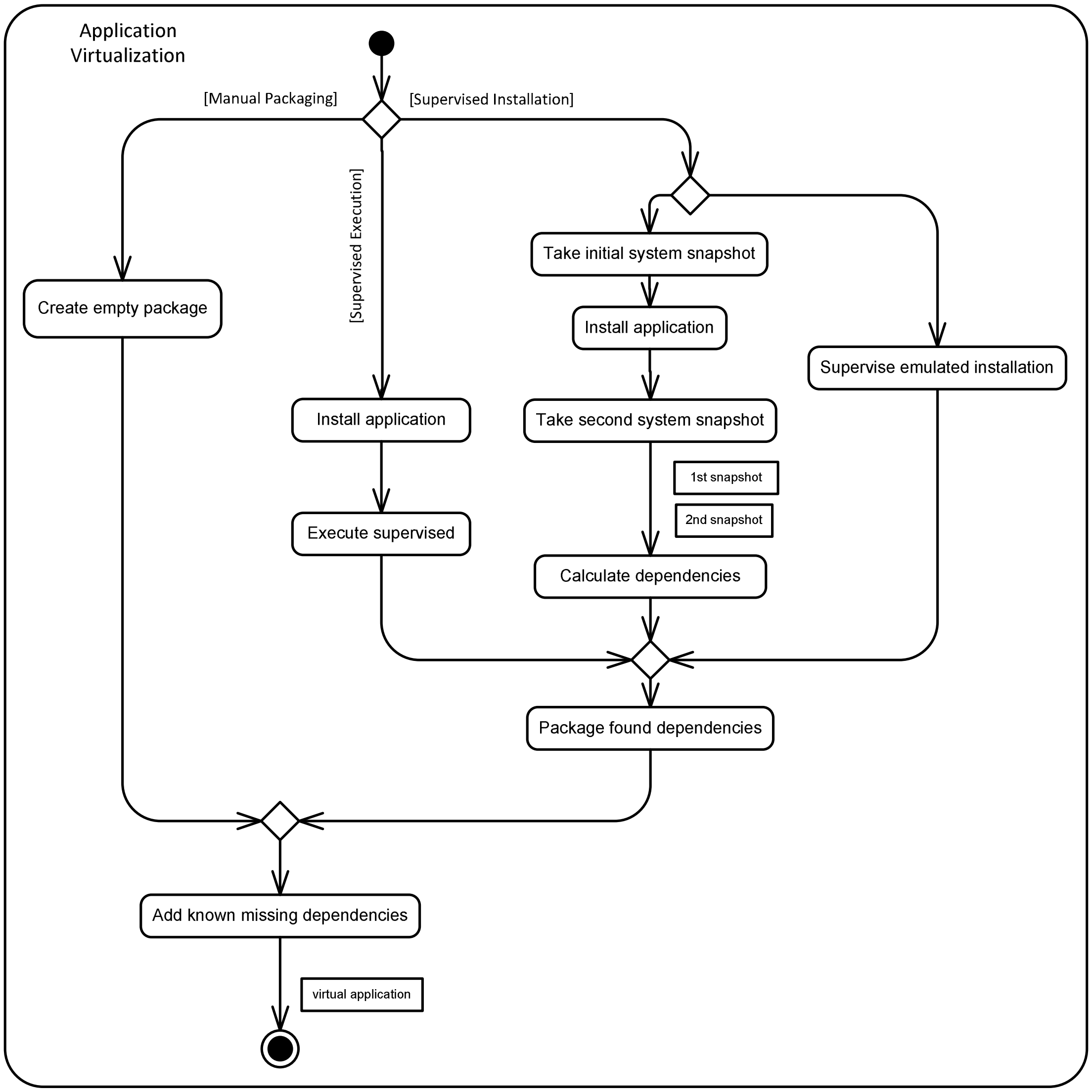

Figure 21: Alternatives for Virtual Application Creation

The result of our comparison is that application preparation for VM synthesis is easier than for application virtualization. However, if the developer knows all of the application's dependencies, preparation for application virtualization is faster than for VM synthesis because computation of disk and memory snapshots for the overlays is slow.

7.4.5 Operation Overhead

For running an application server in the cloudlet implementation that we introduced in Section 6, the server is embedded into the runtime environment of an application virtualization, which in itself runs on a VM. Figure 19 depicts these layers. The application virtualization runtime environment intercepts all of the application's system calls and replaces them with system calls that address resources inside of the virtualized package, rather than resources that reside outside in the operating system's file system. Consequently, the number of the application context switches is three times higher than in normal execution. The first switch occurs with the first system call, the second when the kernel switches to the virtualization runtime, and the third to switch to the kernel for executing the modified system call. The authors of CDE measured the runtime performance impact of their virtualized applications and found a slowdown rate ranging from 0% to 28%. Due to system call frequency, CPU-bound applications had the smallest slowdowns and I/O-intensive tasks had the largest slowdowns [Guo 2011, p. 13].

The application virtualization runtime environment influences execution performance and so does the hardware virtualization layer. Hardware virtualization enables execution of the virtualized application on a VM rather than on the native OS. Hosting a guest OS within a VM causes both CPU overhead as well as memory overhead, unlike running the OS directly on the physical hardware [Larabel 2009].

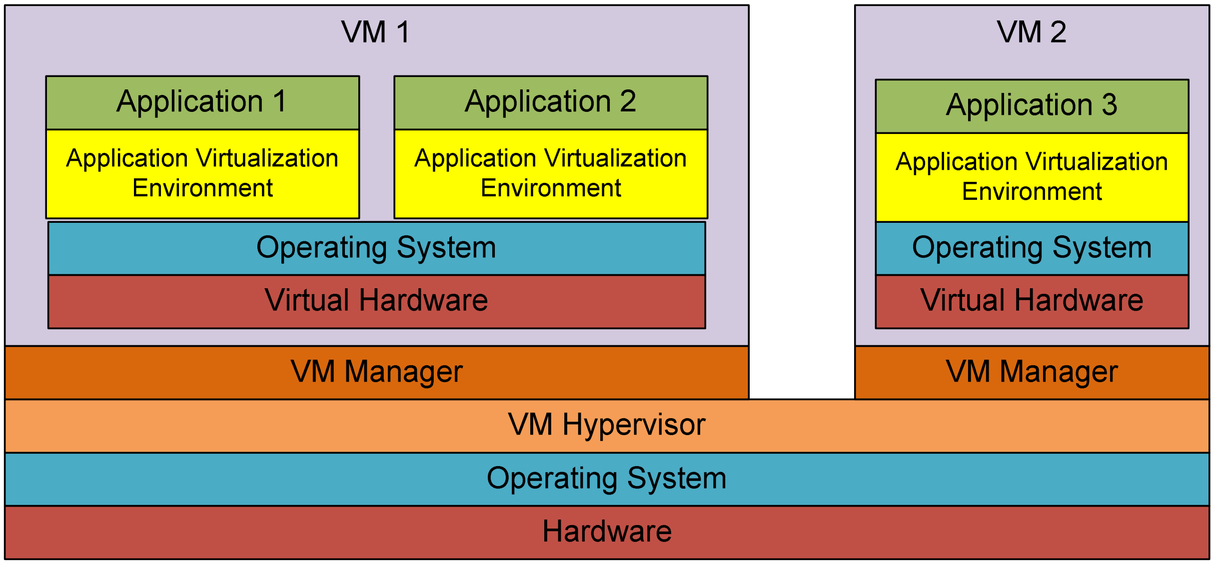

Figure 22: Application Virtualization Layered Architecture

The VM-synthesis-based implementation suffers from that same overhead that is caused by running applications on a virtual rather than a physical machine. However, it does not experience the additional overhead caused by runtime environments such as CDE or Cameyo.

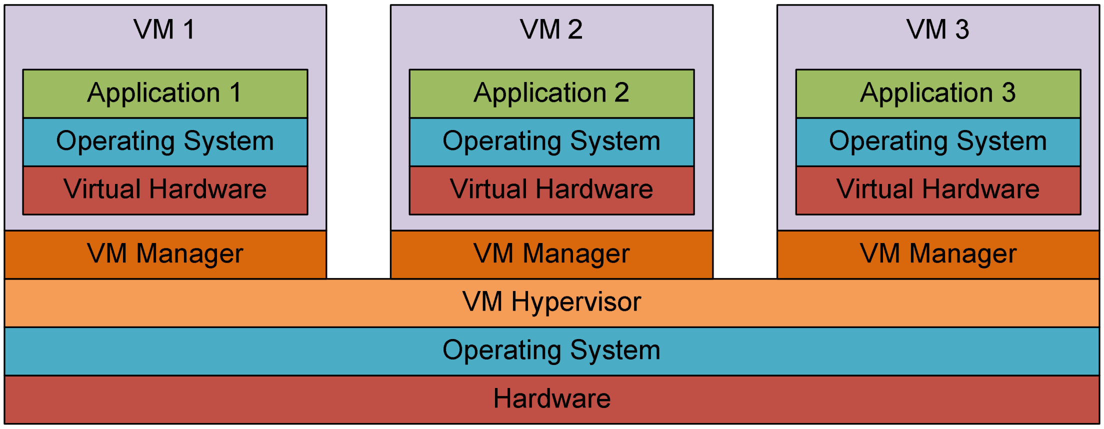

Figure 23: VM Synthesis Layered Architecture

Given the cloudlet server implementation described by Simanta [Simanta 2012], VM synthesis potentially requires more running VMs than the application-virtualization-based solution because it hosts one VM per application. In contrast, the application virtualization cloudlet server maps applications of the same OS family onto one VM (see Figure 22 as compared to Figure 23). Providing a complete OS such as Ubuntu for only a single application includes system functionality that is not required for the specific application and adds to the computational overhead. However, an alternative VM synthesis implementation may include multiple applications in a single overlay, and an alternative application virtualization implementation may use one VM per virtual application.[2]

7.4.6 Isolation and Security

Hardware virtualization adds an additional layer between the physical hardware and the guest OS. This layer is the VM hypervisor which either runs on the host operating system—a *type 2* hypervisor—or directly on top of the physical hardware—a *type 1* hypervisor. In both cases, the VM on which the guest OS is running is isolated from the OS that runs natively on the physical hardware. Therefore, if the guest OS is compromised or malfunctions, the host OS remains unaffected. Consequently, a virtualization environment is more secure because it protects the host OS from damage. Both the application virtualization strategy and the VM synthesis strategy are VM-based and therefore have the isolation benefit.

Another concern related to security is isolation between applications. Comparing application virtualization and VM synthesis, the degree of isolation between applications differs.

VM synthesis hosts one application per VM, thus providing high isolation. One VM cannot affect the other by design, so a failed VM is only a risk to the one application that it hosts. Nevertheless, a potential security risk remains because VMs on one machine share the same physical resources. If a compromised VM succeeds in carrying out a denial-of-service attack, thus blocking the physical hardware, or if it intrudes on the commonly used network, the other VMs will also be harmed [Chen 2012, p. 6].

The application virtualization cloudlet solution runs applications that need the same operating system family on the same VM. This requires isolation mechanisms because applications share the memory, disk, and other system utilities. Basic isolation is provided by the virtualization runtime environment; each application is embedded into a sandbox that isolates it not only from the guest OS but also from other applications. CDE and Cameyo both offer sandboxing techniques to virtualize system resources such as the file system. Sandboxing uses a lower degree of isolation and is therefore not as secure as separation at the VM level. CDE runs the packaged application within a *chroot jail* [FreeBSD 1993], thus preventing it to access files outside its package. This sandbox is vulnerable, however, to attacks that break the isolation mechanism [Simes 2002].

Consequently, VM synthesis offers better isolation between applications than application virtualization. Both strategies equally isolate the system that runs the offloaded application from the underlying hardware and the native OS (type 2 hypervisor).

---

[2] VM synthesis overlays with multiple applications reduce VM overhead at the cost of reduced dynamics. Application virtualization with one VM per application increases VM overhead but is reasonable for better isolation (see Section 7.4.6).

7.4.7 Summarized Comparison of VM Synthesis and Application Virtualization

Table 5: Qualitative Comparison of VM Synthesis and Application Virtualization

	VM Synthesis	**Application Virtualization**
Cloudlet Coupling	Exact base VM	OS family
System Patchability	Complicated	Supported
Application Range	Broad	Limited
Correct Operation	Guaranteed (system replica)	Guaranteed if no missing dependencies
Application Preparation	Simple, slow	Potentially complicated
Operation Overhead	Many VMs	More system calls
Isolation between Applications	Separate VMs	Sandboxing

8 Related Work

Many have researched the idea of leveraging external resources to augment the capabilities of resource-limited mobile devices, termed as cyber foraging [Satyanarayanan 2001]. Multiple cyber-foraging systems have been developed, which differ in terms of the strategy that they use to leverage remote resources.

One strategy is to partition code into segments that run either on the mobile device or on a remote machine. Manual partitioning requires the developer to explicitly mark code to be executed remotely and possibly declare execution profiles. Based on analysis of the impact on performance metrics, the partitioning algorithm selects the optimal profile, which then determines when to offload code to the remote machine. Examples of such cyber-foraging systems are Spectra [Flinn 2001], Chroma [Balan 2002, 2003, 2007], MAUI [Cuervo 2010] and Scavenger [Kristensen 2010].

CloneCloud follows the same code partitioning principle but automatically partitions code at the thread level without need for manual code annotation [Chun 2011]. Remote execution takes place on a clone of the original device, which is encapsulated inside a VM on the remote machine.

Another cyber-foraging strategy is to offload an entire application. Goyal and Carter enable a mobile device to trigger remote download and installation of applications on an external VM [Goyal 2004]. This approach is related closely to the work presented in this technical note. A main difference is that the work described in this note uses cloudlets as offload sites, which do not rely on internet access. In addition, the cloudlet is not altered via remote installations that may lead to dependency conflicts or overloaded systems. Application virtualization eliminates the need for durable installation.

The cloudlet architecture used in the cyber-foraging implementation presented in this technical note has been described by Satyanarayanan and colleagues and by Ha and colleagues [Satyanarayanan 2001, Ha 2011]. Offloading takes place by establishing a VM on the external machine that includes an application that carries out resource-intensive work on behalf of the mobile device. In order to establish this VM efficiently, a strategy called VM synthesis is implemented [Satyanarayanan 2001, Ha 2011, Simanta 2012]. The mobile device carries an application overlay that enables the cloudlet to reconstruct the entire VM. One scenario for a VM synthesis cloudlet system is cyber foraging in hostile environments that are characterized by the lack of reliable wide-area networks [Ha 2011]. The work in this technical note also focuses on providing external resources to mobile devices in hostile environments. However, instead of using a VM synthesis strategy, our work explores the applicability of application virtualization as a strategy for cyber foraging. We implemented an architecture for application virtualization, and described it and compared it to VM synthesis in this technical note.

9 Limitations and Future Work

All cyber-foraging strategies have pros and cons. Application virtualization is not the exception. Application virtualization as a concept requires all application dependencies to be identified and packaged. Because it is impossible to detect automatically all dependencies, human knowledge of the application is required. This is especially true for applications that have a plug-in architecture. The application virtualization tools used in this technical note allow for the manual addition of dependencies in order to create a complete virtual package. Future work in this area should focus on facilitating the process of creating complete packages. A possible approach is to declare explicitly dependencies in a document similar to a manifest file. However, declaring folders or files manually is cumbersome, in which case a tool may help by suggesting typically used components for inclusion. In addition, application virtualization does not have portability benefits if applications rely on specific hardware or device drivers. Requiring a very particular environment is against the idea of general-purpose cloudlets.

Specific limitations of the implementation architecture presented in this technical note, and recommendations for future work, include the following:

- The implementation architecture presented in this technical note does not allow application servers running on the same cloudlet to have the same port number. However, because sandboxed applications can share common resources such as port numbers, they may conflict with each other. This means that some form of virtualization or redirection must be introduced into the architecture that decouples fixed port numbers from the actual ports provided by the cloudlet. Future work should analyze the overhead of isolating each application into its own VM, as is done in VM synthesis.
- A real-world cloudlet solution must satisfy security requirements, which we have not discussed in this technical note. Therefore, the implementation should be extended with trust establishment mechanisms between mobile devices and cloudlets.
- The mechanism for discovering a cloudlet that fits the application's requirements is rather primitive because it assumes that the cloudlet and mobile device use the same keywords for properties. As a consequence, the declaration of cloudlet capabilities and application demands must be formalized in future work.
- An important aspect of mobility is the ability to change cloudlets as these become out of range of the mobile devices or better cloudlets come into proximity. This requires live migration, that is, resuming the halted application on another cloudlet while preserving computational state and minimizing downtimes. Migration of virtualized applications may be a topic of future work.

Although application virtualization can be seen as an alternative to VM synthesis, this does not imply that these two strategies disqualify each other. On the contrary, they may complement one another and together increase the possibility of cyber foraging. A cloudlet may support both strategies with application virtualization being the preferred one because it is faster. VM synthesis in this case could be a fallback strategy in case there is not a match for the application or there is already a valid VM for the application. The combination of VM synthesis and application virtualization requires further exploration.

10 Conclusions

Cyber foraging, that is, offloading of resource-intensive tasks to external resource-rich machines enables mobile devices to provide acceptable performance for costly computations. At the same time the mobile device saves energy, which leads to longer battery life.

In this technical note, we have focused on cyber foraging in hostile environments where reliable networks are not guaranteed. A connection to a distant cloud or data center especially cannot be assumed. The role of the code offload site is played by cloudlets instead, which are machines in close proximity that make their resources available to mobile devices.

Our cyber-foraging mechanism is based on the client-server principle; thereby the application client runs on the mobile device and the application server on the cloudlet. Before utilizing the cloudlet, the mobile device must deploy its application server on the cloudlet. Related work that uses VM synthesis [Satyanarayanan 2001, Ha 2011, Simanta 2012] for cloudlet provisioning has been the driver for this technical note.

In this technical note, we provided an outline of the difficulties of application portability, followed by an introduction to application virtualization. In the context of this work, a cloudlet-based cyber-foraging system that uses application virtualization was implemented. We introduced this implementation by presenting an architectural overview and discussing selected implementation decisions. We evaluated the implementation and compared the achieved results and characteristics with the VM synthesis strategy. Finally, we identified the limitations of this work and presented some ideas for future work.

To summarize the evaluation, application virtualization has an advantage over VM synthesis in terms of deployment phase performance. The two metrics for our definition of performance are application deployment time and energy consumption for application deployment. We have shown that these two metrics have a linear dependency with the amount of data that must be transmitted to the cloudlet. This observation is true for both application virtualization and VM synthesis. The better performance of application virtualization results from significantly smaller file sizes. Another benefit of application virtualization is the loose coupling between an application and its required cloudlet environment. While having a member of an appropriate operating system family is sufficient for application virtualization, VM synthesis requires the cloudlet to provide a binary equal operating system. Therefore, application virtualization facilitates the provision of suitable cloudlet environments and allows for system patches without invalidation of the relationship between cloudlet and application.

In other aspects, application virtualization falls behind VM synthesis. Hardware-specific dependencies such as device drivers cannot be virtualized. VM synthesis does not suffer from this limitation because all hardware is virtualized. Application virtualization requires careful manual dependency management to guarantee an application’s correct operation on a cloudlet. VM synthesis requires no knowledge other than what is necessary for an ordinary installation process.

Nonetheless, application virtualization is a promising strategy for cyber foraging in resource-constrained environments because of it is a lightweight approach that offers high portability. It can also be viewed as a complement to VM synthesis in a combined cyber-foraging model. Future work may build on the implementation in this technical note and try to overcome some of its shortcomings.

References

URLs are valid as of the publication date of this document.

[Apple 2012]
Apple Corporation. *iOS*, Siri.
http://www.apple.com/iphone/features/siri.html (2013).

[Balan 2002]
Balan, Rajesh, et al. 2002. "The Case for Cyber Foraging," 87-92. *Proceedings of the 10th ACM SIGOPS European Workshop*. Saint-Emilion, France, July 2002. ACM, 2002.

[Balan 2003]
Balan, Rajesh, et al. "Tactics-Based Remote Execution for Mobile Computing," 273-286. *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys '03)*. San Francisco, CA, U.S.A., May 2003. ACM, 2003.

[Balan 2007]
Balan, Rajesh Krishna, et al. "Simplifying Cyber Foraging for Mobile Devices," 272-285. *Proceedings of the 5th International Conference on Mobile Systems Applications (MobiSys '07).* San Juan, Puerto Rico, June 2007. ACM, 2007.

[Cameyo 2002]
Cameyo User Guide, Version 2. Cameyo, 2002.
http://cameyo.com/doc/CameyoManual.pdf

[Cameyo 2012]
Latest Cameyo 2. Cameyo Blog, 04 25, 2012.
http://cameyoco.blogspot.com/2012/04/latest-cameyo-2.html

[Chen 2012]
Chen, Quian, et al. 2012. "On State of The Art in Virtual Machine Security," 1-6. *Southeastcon, 2012 Proceedings of IEEE*. Orlando, Florida, U.S.A., March 2012. IEEE, 2012.

[Chinnici 2007]
Chinnicci, Roberto, et al. 2007. "Web Services Description Language (WSDL) Version 2.0." *World Wide Web Consortium (W3C)*, 2007. http://www.w3.org/TR/wsdl20/

[Chun 2011]
Chun, B., et al. "CloneCloud: Elastic Execution Between Mobile Device and Cloud," 301-314. *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. Salzburg, Austria, April 2011. ACM, 2011.

[Crockford 2006]
Crockford, Douglas. RFC 4627 *The Application/json Media Type for JavaScript Object Notation (JSON).* Network Working Group, 2006. http://tools.ietf.org/html/rfc4627

[CMU 2011]
Carnegie Mellon University Personal Robotics Lab. *MOPED: Object Recognition and Pose Estimation for Manipulation*. 2011. http://personalrobotics.ri.cmu.edu/projects/moped.php

[CMU 2012]
Carnegie Mellon University. “Sphinx-4: A Speech Recognizer Written Entirely in the Java Programming Language.” 2012. http://cmusphinx.sourceforge.net/sphinx4/

[Cuervo 2010]
Cuervo, Eduardo, et al. “MAUI: Making Smartphones Last Longer with Code Offload.” *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (*MobiSys ’10*). San Francisco, CA, USA: June 2010. ACM, 2010.

[Eclipse 2012]
Eclipse “jetty://” http://www.eclipse.org/jetty/ (2013).

Fielding 2000]
Fielding, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD diss., University of California, 2000.

[Flinn 2001]
Flinn, Jason S., Narayanan, Dushyanth, & Satyanarayanan, Mahadev. “Self-Tuned Remote Execution for Pervasive Computing.” *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Schloss Elmau, Germany, May 2001. IEEE, 2001.

[FreeBSD 1993]
FreeBSD. *FreeBSD System Calls Manual*, chroot (2) 1993.
http://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2

[Gamma 1995]
Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. *Design Patterns,* pp. 151-161. Addison-Wesley, 1995.

[Google 2012a]
Google Inc. “Google Goggles.” 2012. http://www.google.com/mobile/goggles/

[Google 2012b]
Google Inc. “Google Maps API Web Services.” 2012.
https://developers.google.com/maps/documentation/webservices/

[Goyal 2004]
Goyal, Sachin & Carter, John. “A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices.” *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems (WMCSA 2004)*. Lake District National Park, U.K., December 2004. IEEE, 2004.

[Guo 2011]
Guo, Philip J. & Engler, Dawson. “CDE: Using System Call Interposition to Automatically Create Portable Software Packages,” 21-21. *Proceedings of the 2011 USENIX Annual Technical Conference*. Portland, OR, U.S.A., June 2011. USENIX Association, 2011.

[Ha 2011]
Ha, Kiryong, Lewis, Grace, Simanta, Soumya, & Satyanarayanan, Mahadev. *Cloud Offload in Hostile Environments (CMU-CS-11-146)*. Carnegie Mellon University, 2011.

[Hunt 1999]
Hunt, Galen C. & Scott, Michael L. "The Coign Automatic Distributed Partitioning System," 187-200. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. New Orleans, Louisiana, U.S.A., February 1999. USENIX Association, 1999.

[Infographic 2012]
Infographic. "Smartphone Users Around the World - Statistics and Facts." 2012. http://www.go-gulf.com/blog/smartphone

[Itseez 2012]
Itseez. "OpenCV (Open Source Computer Vision)." 2012. http://opencv.org/

[Jackson 2002]
Jackson, Ian & Schwarz, Christian. 2012. "Debian Policy Manual: Chapter 7 - Declaring Relationships Between Packages." 2012. http://www.debian.org/doc/debian-policy/ch-relationships.html

[Kristensen 2010]
Kristensen, Mads Darø. *Empowering Mobile Devices Through Cyber Foraging.* Aarhus University, Department of Computer Science, 2010.

[KVM 2011]
KVM "Setting Guest Network." 2011. http://www.linux-kvm.org/page/Networking

[Larabel 2009]
Larabel, Michael. "Intel Core i7 Virtualization Performance." 2009. http://www.phoronix.com/scan.php?page=article&item=intel_corei7_virt&num=1

[Microsoft 1993]
Microsoft Corporation. "Component Object Model (COM)." 1993. http://msdn.microsoft.com/en-us/library/ms680573(v=vs.85)

[Mobithink 2011]
Mobithink. "2011 Handset and Smartphone Sales Statistics Worldwide: the Big Picture." 2012. http://mobithinking.com/blog/2011-handset-and-smartphone-sales-big-picture

[Monsoon 2008]
Monsoon Solutions Inc. "Power Monitor." http://www.msoon.com/LabEquipment/PowerMonitor/ (2008).

[Morris 2011]
Morris, Edwin. "A New Approach for Handheld Devices in the Military." SEI Blog. 2011. http://blog.sei.cmu.edu/post.cfm/a-new-approach-for-handheld-devices-in-the-military

[OMG 2011]
Object Management Group (OMG). “CORBA Interface Definition Language Specification, Version 3.5.” 2011.

[Linux 2013]
Linux. “ptrace (2) - Linux man page.” http://linux.die.net/man/2/ptrace (Accessed: 2 27, 2013).

[RPM 2005]
RPM. “RPM Packager Documentation: Dependencies.” 2005.
http://www.rpm.org/wiki/PackagerDocs/Dependencies

[Satyanarayanan 2001]
Satyanarayanan, Mahadev. 2001. “Pervasive Computing: Vision and Challenges. ” *IEEE Personal Communications.* August 2001, pp. 10-17.

[Satyanarayanan 2009]
Satyanarayanan, Mahadev, Bahl, P., Caceres, R., & Davies, N. “The Case for VM-Based Cloudlets in Mobile Computing.” *IEEE CS Pervasive Computing 8*, 4. (October 2009):14-23.

[Simanta 2012]
Simanta, Soumya, et al. 2012. *Cloud Computing at the Tactical Edge (CMU/SEI-2012-TN-015)*. Software Engineering Institute, Carnegie Mellon University, 2012.
http://www.sei.cmu.edu/library/abstracts/reports/12tn015.cfm

[Simes 2002]
Simes. “How to break out of a chroot() jail.” 2002.
http://www.bpfh.net/simes/computing/chroot-break.html

[van Hoff 2011]
van Hoff, Arthur, Blair, Rich & Frisch, Pierre. “JmDNS.” 2011. http://jmdns.sourceforge.net/

[Wang 2005]
Wang, Xiaoyun &Yu, Hongbo. “How to Break MD5 and Other Hash Functions.” *Advances in Cryptology – EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Aarhus, Denmark, May 2005. *Lecture Notes in Computer Science*, Springer, 2005.

[Wolbach 2008]
Wolbach, Adam. *Improving the Deployability of Diamond*. Carnegie Mellon University, School of Computer Science. 2008.
http://reports-archive.adm.cs.cmu.edu/anon/2008/abstracts/08-158.html

[Zeroconf 2012]
IETF Zeroconf Working Group. “Zero Configuration Networking (Zeroconf).” (Accessed 02 27, 2013.) http://www.zeroconf.org/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** May 2013	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE**

Application Virtualization as a Strategy for Cyber Foraging in Resource-Constrained Environments

5. **FUNDING NUMBERS**

FA8721-05-C-0003

6. **AUTHOR(S)**

Dominik Messinger , Grace Lewis

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**

CMU/SEI-2013-TN-007

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

AFLCMC/PZE/Hanscom

Enterprise Acquisition Division

20 Schilling Circle

Building 1305

Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**

n/a

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**

Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

Modern mobile devices create new opportunities to interact with their surrounding environment, but their computational power and battery capacity is limited. Code offloading to external servers located in clouds or data centers can help overcome these limitations. However, in hostile environments, it is not possible to guarantee reliable networks, and thus stable cloud accessibility is not available. Cyber foraging is a technique for offloading resource-intensive tasks from mobile devices to resource-rich surrogate machines in close wireless proximity. One type of such surrogate machines is a cloudlet—a generic server that runs one or more virtual machines (VMs) located in single-hop distance to the mobile device. Cloudlet-based cyber foraging can compensate for missing cloud access in the context of hostile environments. One strategy for cloudlet provisioning is VM synthesis. Unfortunately, it is time consuming and battery draining due to large file transfers. This technical note explores application virtualization as a more lightweight alternative to VM synthesis for cloudlet provisioning. A corresponding implementation is presented and evaluated. A quantitative analysis describes performance results in terms of time and energy consumption; a qualitative analysis compares implementation characteristics to VM synthesis. The evaluation shows that application virtualization is a valid strategy for cyber foraging in hostile environments.

14. **SUBJECT TERMS**

application virtualization, cloudlet, cyber foraging

15. **NUMBER OF PAGES**

63

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102