Karlsruher Institut für Technologie
Fakultät für Informatik

Carnegie Mellon University
School of Computer Science

Bachelor Thesis

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

Dominik Messinger

November 12, 2012

Supervisors:

Prof. Dr. Ralf Reussner

Grace Lewis

Acknowledgements

First of all, I would like to thank Grace Lewis from the Carnegie Mellon Software Engineering Institute (SEI) and Prof. Ralf Reussner from the Karlsruhe Institute of Technology (KIT) whose collaboration made this thesis possible in the first place. Thank you for all advice and valuable feedback. Next, I would like to thank the members of the cloudlet working group at CMU whose lively discussions and interesting ideas contributed to my work and gave me insight into modern mobile computing research. Finally, I would like to thank the interACT office and the Baden-Württemberg-Stiftung. Their scholarship enabled me to compile my Bachelor thesis in this great environment of Carnegie Mellon University and SEI.

Declaration of Authorship

Ich versichere, die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht zu haben. Die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der zu diesem Zeitpunkt gültigen Fassung habe ich beachtet.

Karlsruhe, den 12. November 2012

Dominik Messinger

Zusammenfassung

Moderne Mobilgeräte gewähren ihren Nutzern neue Interaktionsmöglichkeiten mit der sie umgebenden Umwelt, die jedoch durch begrenzte Rechenkapazität und Akkulaufzeit eingeschränkt werden. Die Auslagerung von Arbeit in eine Cloud oder ein Rechenzentrum mittels Code Offloading kann dieser Einschränkung begegnen, indem dem Mobilgerät notwendige Ressourcen für rechenaufwändige Anwendungen wie Spracherkennung, Bildverarbeitung und Decision-Making zur Verfügung gestellt werden. Nichtsdestotrotz findet diese Technik ihre Grenzen bei der Anwendung in sogenannten *Hostile Environments*, die keine verlässliche Netzwerkverbindung und folglich keine stabile Cloud-Konnektivität zulassen. Als Beispiele für solche Umgebungen seien Schauplätze von Militäreinsätzen oder von Naturkatastrophen heimgesuchte Landschaften genannt.

Cyber Foraging ist eine Code-Offload-Technik, bei der ressourcenintensive Arbeit vom Mobilgerät an durch WLAN erreichbare, ressourcenstarke Rechner in der näheren Umgebung ausgelagert wird. Eine Variante dieser dedizierten Rechner sind sogenannte Cloudlets: generische Server in Single-Hop-Distanz zum Mobilgerät, auf denen ein oder mehrere virtuelle Maschinen (VMs) die zu verrichtende Arbeit ausführen. Cloudlet-basiertes Cyber Foraging kann die fehlende Verbindung zu einer Cloud im Kontext von Hostile Environments kompensieren. Eine Cloudlet-Strategie ist VM-Synthese; wenngleich deren Anwendbarkeit bereits in verwandter Arbeit gezeigt wurde, gestaltet sich der eingesetzte Deployment-Prozess zeitaufwändig und energieintensiv auf Grund großer Datentransfers.

In der vorliegenden Bachelorarbeit wird die Anwendung von Applikationsvirtualisierung zur Cloudlet-Bereitstellung als eine leichtgewichtigere Alternative zu VM-Synthese untersucht. Eine entsprechende Implementierung wird vorgestellt und evaluiert. In einer quantitativen Analyse werden Performance-Ergebnisse bezogen auf Zeit- und Energieverbrauch beschrieben; in einer qualitativen Analyse werden Implementierungsmerkmale mit VM Synthese verglichen. Die Evaluation zeigt schlussendlich, dass Applikationsvirtualisierung eine zulässige Strategie für Cyber Foraging in Hostile Environments darstellt.

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 that are located in clouds or data centers can help to overcome these limitations and provide the necessary resources for computation-intensive tasks such as speech recognition, image processing or decision-making. However, in hostile environments, such as theaters of military operations or natural disaster recovery areas, reliable networks cannot be guaranteed 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 are cloudlets, which are generic servers that run one or more virtual machines (VMs) and are 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. A particular strategy for cloudlets is VM synthesis. While its general applicability has been shown in related work, the deployment process is time-consuming and battery-draining due to large file transfers.

This thesis explores the applicability of 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.

Content

1 Introduction

1.1 Background and Motivation

Mobile Computing has arrived at the heart of our society and its impact on our everyday lives is steadily growing. With smartphones having had its commercial breakthrough in recent years [1] [2], 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 to share visual impressions immediately and aid to scan and process information in the direct environment.

The vision of ubiquitous computing that provides additional information that exceeds our natural recognition capabilities and thereby augments human perception is already a reality. However, there are still limitations to mobile devices because they tend to fail to meet the needs for 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 amongst the most desired services for mobile devices [3]. To deal with the issue of resource limitation, techniques have been developed that offer mobile devices access to more 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 are the natural language processing software *Siri,* which needs access to the Apple Cloud [4], or *Google Goggles* communicating with Google servers for providing image recognition [5]. To use cloud resources, the mobile device has to be connected to the Internet in order to establish a connection to the cloud services. Although this seems to be an appropriate solution for many use cases, there are some shortcomings to cloud usage.

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. It has been 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 [3]. 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, e.g. augmented reality software which has to process and display information in real-time. Therefore, data centers need to be in close proximity depending on the type of service that is demanded by the mobile device. For example, highly responsive augmented reality applications need reasonably-high frame rates in order to provide a fair usage experience (c.f. p.17 in [3]). As long as services are bound to predetermined servers, these have to be close enough to the mobile device in order 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 or in the context of disaster recovery. An example of a resource-constrained environment with such conditions has been described as *hostile environments* by Ha et al. [6]. 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, e.g. as a consequence of earthquakes or war actions. They can also be compromised by opponents that intrude into the network and carry out attacks. Considering such cyber war attacks, Ha et al. assume that even the Internet may one day become a hostile environment in the mentioned sense [6]. The Department of Defense has shown a strong interest in equipping soldiers with handheld devices to enhance their operational abilities [7]. 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 et al. propose a solution for code offloading, i.e. transferring resource-intensive tasks to stronger external machines, in hostile environments [6]. It utilizes cyber foraging techniques in single-hop networks and virtualization technology. The term cyber foraging – first introduced in [8] – describes the technique of code offloading to nearby surrogate machines, so-called *cloudlets*. These cloudlets are in close proximity to the mobile device and can be accessed via a single-hop network. Such a setup differs from far distant clouds and 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 wide-area networks (p.4) [6].

In order to enable cyber foraging, a mobile application is divided into a client running on the mobile device and a server running on the cloudlet. The server has to be deployed on the cloudlet before it can be accessed by the mobile client. This deployment is accomplished via *VM synthesis*: the cloudlet holds 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 the server has been installed. 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.

A reference implementation is presented in [9]; Figure 1 shows the process of overlay creation.

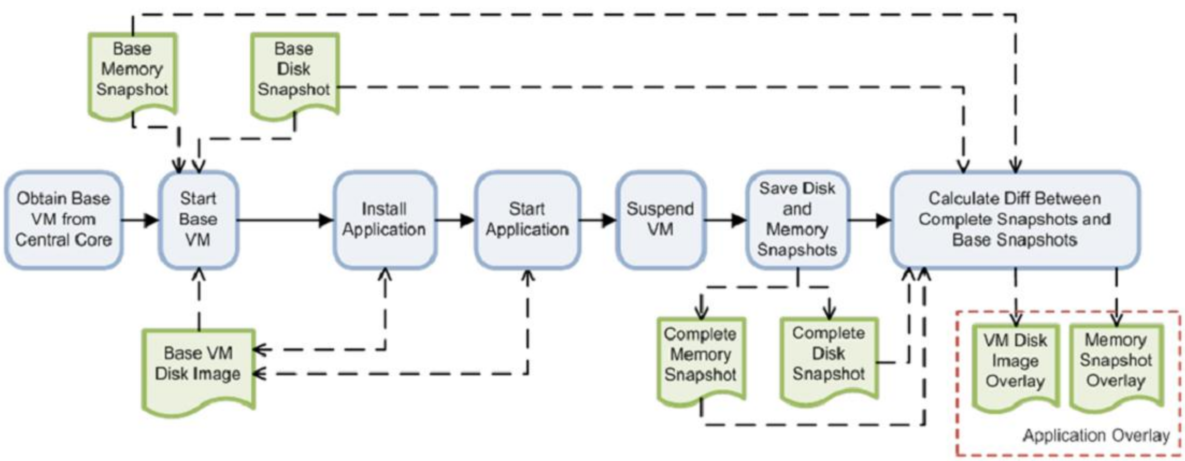

Figure 1: VM overlay creation (p.15) [9]

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 (cf. p. 19) [9]. Regarding flexibility, VM synthesis requires to match a VM overlay with the same base VM that was using 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.

1.2 Goal and Structure of this Thesis

The subject of this thesis 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 the context of this thesis, a cyber-foraging framework has been implemented that utilizes application virtualization to provision cloudlets with application servers.

This thesis begins with an introduction to cyber foraging and cloudlets, and continues with a discussion of different techniques for application deployment. An introduction to application virtualization follows. Then, the implementation is presented and evaluated. The evaluation includes a comparison with VM synthesis in regard to its suitability for the operation in hostile environments. Finally, limitations are identified, which 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 amongst the most desired applications for mobile computing [3].

Cyber foraging, as first introduced by Satyanarayanan [8], 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* [8] 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. An illustration is presented in Figure 2.

> *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. No match could be found, so 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.

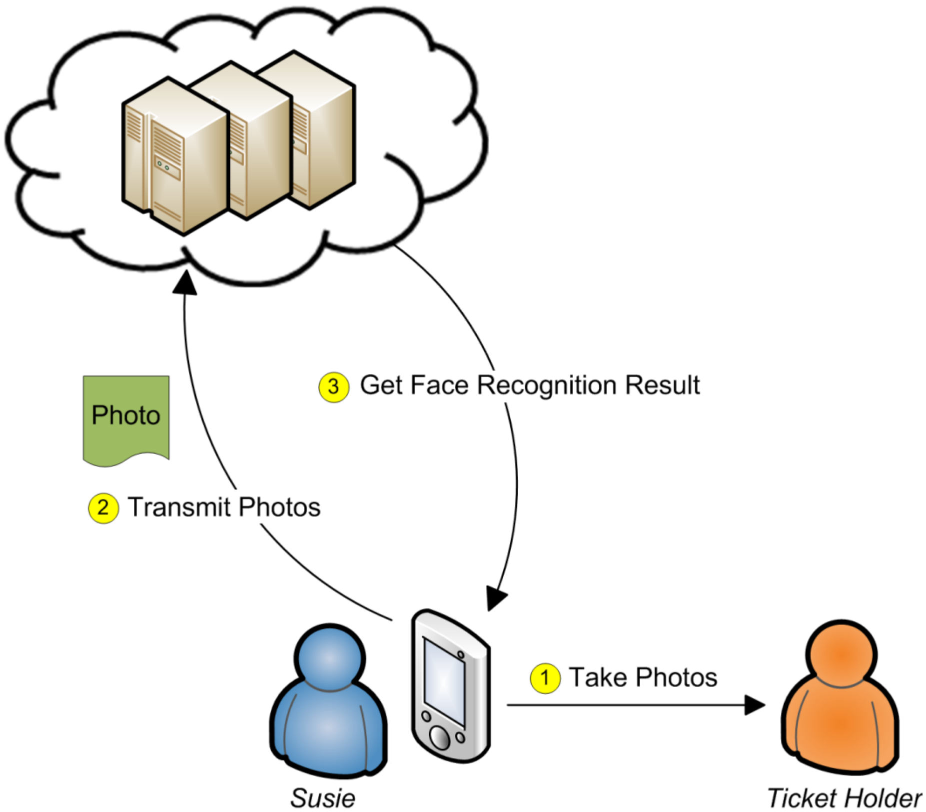

Figure 2: Cyber-foraging scenario "Stadium Security"

2.3 Cyber-Foraging Strategies

In order to relieve the mobile client, the surrogate machine needs to be capable of running offloaded tasks. Thus, a software item that serves this need has to be installed on the surrogate. 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 needs to be offloaded. This is therefore the lightest form of deployment. Examples are already installed web services or components that support remote procedure calls. Such software is accessed through an interface, which is known by the client. This interface is typically defined through an interface description language (e.g. *WSDL 2.0* [10], *CORBA IDL 3.5* [11]).

A real-world cyber-foraging example could be a smartphone app that uses the Google Maps API Web Services [12] for computing the shortest distance between two given locations.

2.3.2 Mobile Code

A different deployment approach has the surrogate execute portions of code that are offloaded from the client. This is different to 2.3.1 because not only data – i.e. service identifier plus arguments – but code is transmitted 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* [13] and its successor *Chroma* [14] 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 [15] [16]. Similar solutions are *Scavenger* [17] and *MAUI* [18], which support code annotations to mark procedures for remote execution.

Designed to let mobile devices take benefit of a cloud, *CloneCloud* [19] offers automatic partitioning at thread level by finding the migration profile with the least migration cost. No source code modification is needed. 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* [20]. Restricted to the Microsoft Component Object Model (*COM*) [21], 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 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 [22] introduced a cyber-foraging strategy that implements application deployment. 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. Different from application deployment, the deployment process is not the installation of an application in a VM, but rather the entire VM is deployed on the surrogate.

Satyanarayanan et al. use a technique called *VM synthesis* to provision the surrogate machine [3]. 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. This technique is described in [23].

2.4 Application Virtualization as a Cyber-Foraging Strategy

The work in this thesis 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 (2.3.3) and VM synthesis to the category of virtual machine deployment (2.3.4).

3 Cloudlets

3.1 Concept

Instead of relying on distant computing clusters such as clouds, Satyanarayanan et al. propose the usage of *cloudlets*. A cloudlet is a computer or computer cluster that serves as a code offloading site for nearby mobile devices [3]. The close one-hop proximity to such a cyber-foraging surrogate avoids possible high latencies (c.f. [3]). 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 (p.15) [3](p.4) [6]. 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 (p.9) [3].

3.2 Architecture

Ha et al. propose a two-level hierarchy with cloudlets at the edge and a cloud at the core [6], cf. Figure 3. The cloudlets serve as offload elements for mobile devices. Connectivity between the cloudlets and the cloud is only required for provisioning — they 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 this architecture was increased computing power and battery life on mobile devices in hostile environments (p.8) [6]. In contrast, Satyanarayanan et al. assumed in [3] that cloudlets would have permanent Internet access (p.14).

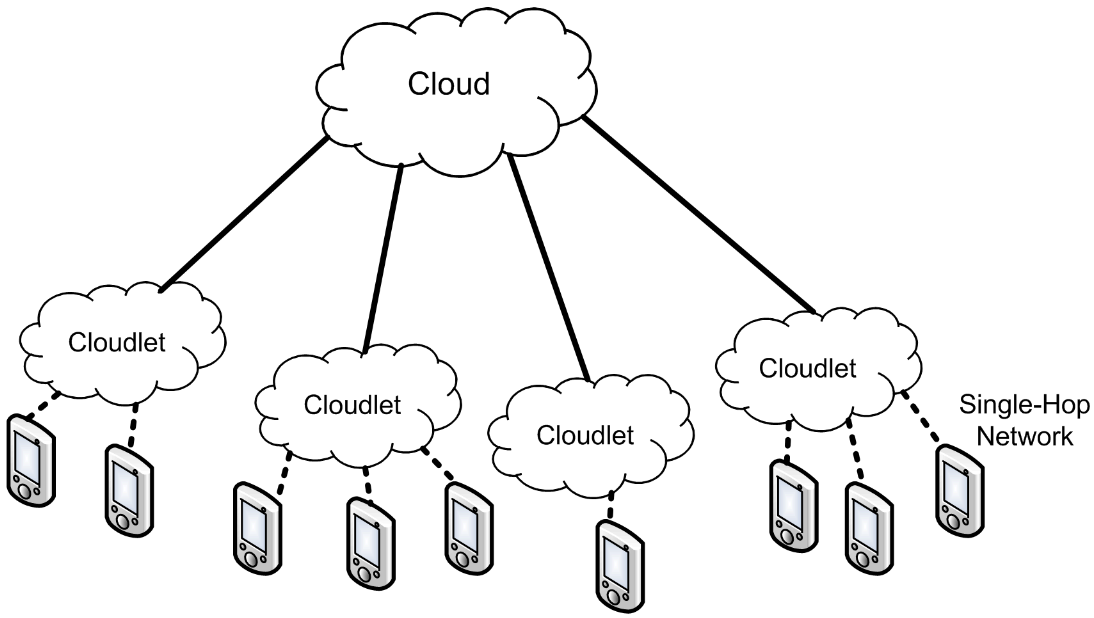

Figure 3: Hierarchical Architecture for Offload to Cloud-connected Cloudlets, based on Ha et al. [6]

3.3 Cloudlet Scenario

The following cyber-foraging scenario is set in a hostile environment and therefore does not rely on the Internet. An illustration is presented in Figure 4.

> *Susie quit her job as a security guard and now works for a NGO that focuses on disaster recovery. Recently, a massive earthquake occurred that caused a high number of casualties and left 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.*

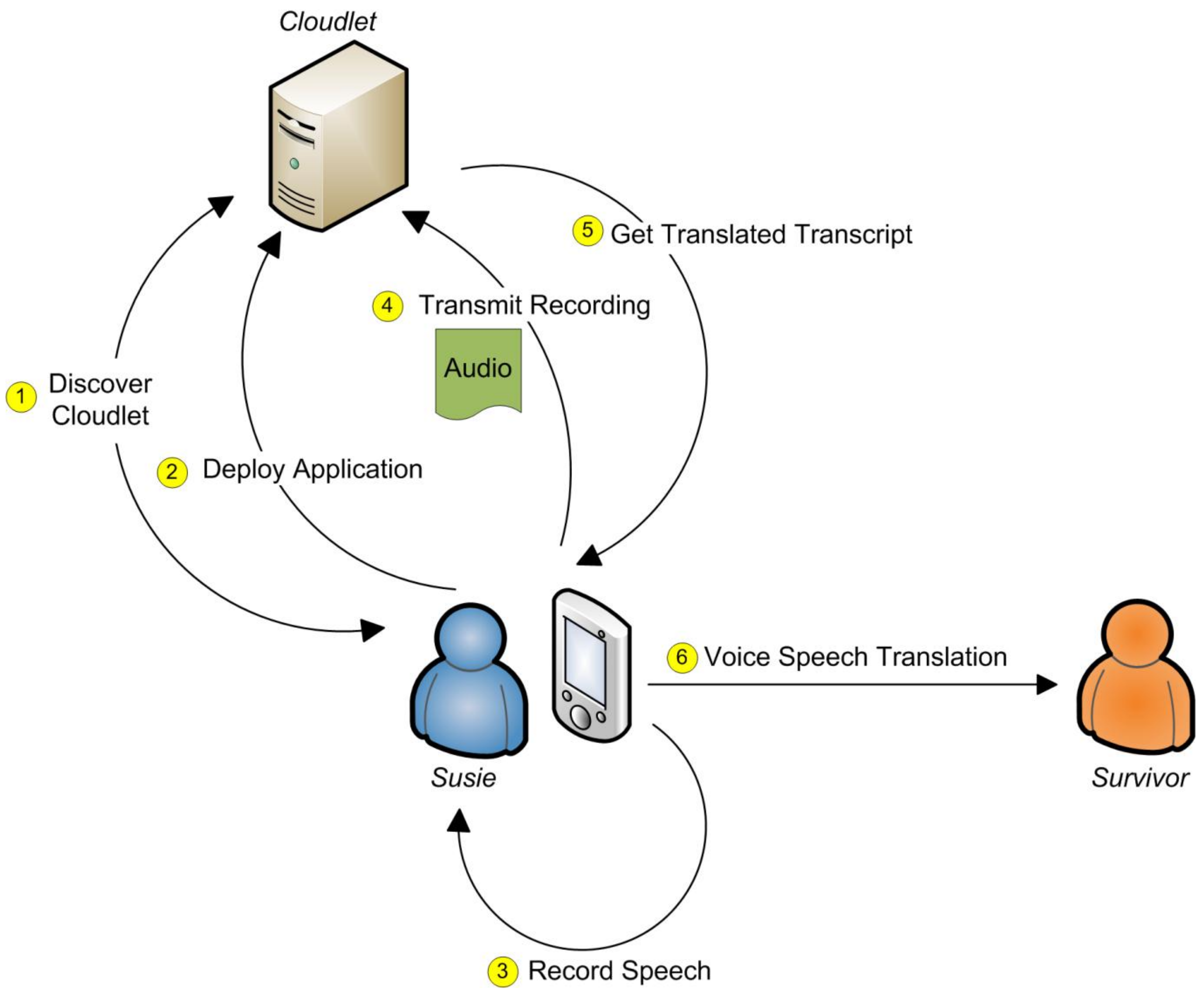

Figure 4: Cloudlet Scenario

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. These phases are shown in Figure 5. Because a cloudlet offers its platform and not pre-installed services, these phases include application deployment.

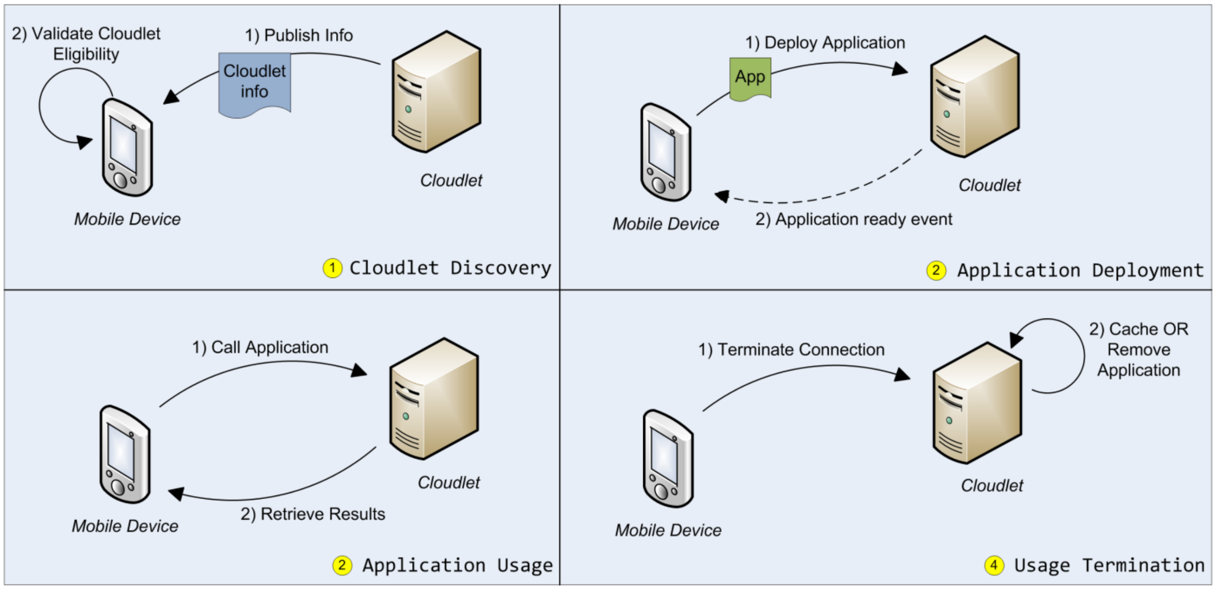

Figure 5: Phases of Cloudlet Interaction

1. Cloudlet Discovery

The mobile device discovers nearby cloudlets and selects the cloudlet that best meets its requirements. This means that each cloudlet has to publish information about itself, which is retrieved through a discovery mechanism.

2. Application Deployment

After the mobile device has selected an appropriate cloudlet, it needs to deploy the application that it would like to have executed remotely. Although the application may already be deployed, the mobile device must be able to deploy it if it is not.

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

Starting from a vision of being easy to deploy, multi-purpose, and transient, the following functional and quality attribute requirements are applicable for cloudlets.

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, e.g. 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, i.e. able to host a variety of applications, and not be limited to 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 needs to 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, all data that is needed for deployment must be provided by the mobile client. An application should be deployable in as many different environments as possible, thus increasing the chance to discover 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 *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 need to be met 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 the application has been designed for, e.g. 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 has to 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 has to 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 either at load time or runtime. Therefore, the library has to 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 that is to be deployed on the cloudlet has to 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 have to 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 (c.f. 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 solved 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 (cf. [24] [25]).

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 which 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 such a way that other applications cannot be deployed. 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 that can be expected 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 can only be achieved if all source code dependencies are available on the target system. Otherwise, already compiled dependencies narrow down the range of possible target systems, regardless of the main application's sources. The assumption of source code availability is invalid for usage of non-open-source software. In this case, shipping the application as a binary does not further limit portability, and in addition avoids the time-consuming compilation on the target system.

4.3 Packaging Dependencies

As a result of 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 sub-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 mentioned in Section 4.1.4, 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 instead send 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 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 the correct version of the standard C library from the Ubuntu 12.04 source system instead, 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 including the compiled code and all library dependencies in one binary. As soon as the linking succeeds, the execution will not fail during load time because of missing dependencies. But 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 needs to 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, it is not a good solution either.

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.

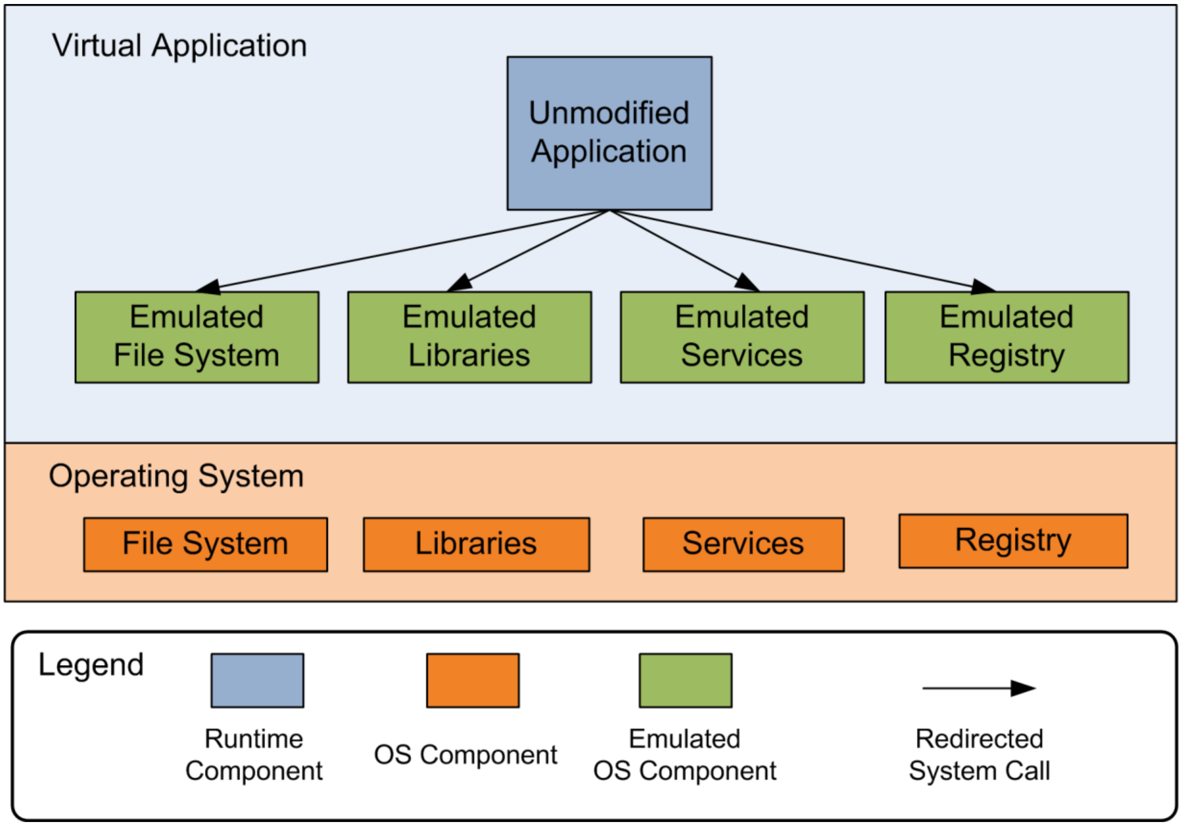

Figure 6: Application Virtualization through OS Component Emulation and System Call Redirection

Application virtualization cannot be used 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 slower 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 in a degree that the approaches 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 thesis 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 were used.

5 Application Virtualization for Cloudlets

5.1 Design Goals

The cloudlet architecture implementation in this thesis uses application virtualization techniques to support the design goals that were 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 has to 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 consequence, the cloudlet should allow regular updates and upgrades to the operating system that runs virtual applications without breaking functionality. All applications that are not too deeply integrated 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. These tools were selected 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 *C*ode, *D*ata and *E*nvironment) is an application virtualizer for Linux developed by Philip J. Guo and Dawson Engler [26].

CDE allows virtualizing applications by monitoring their execution. Through the *ptrace* system call [27], the supervising CDE program finds files that have been accessed during execution and packages them. 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. 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] from several years in the future" (p.4) [26].

The package can be configured to allow access to specified file paths outside its sandbox. CDE does not guarantee to include 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 could theoretically be accessed, every possible control path would have to be examined. This is a non-deterministic problem; solving it would require program behavior to be predicted before execution. Especially for applications that implement a plugin structure and load libraries dynamically 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 newly been accessed by the application. It is also possible to add files manually to the package. This is the preferred way when deeper knowledge about the application exists.

5.2.2 Cameyo

Cameyo [28] is an application virtualizer for Windows. It packages the application and its dependencies into one single executable (.exe) file. Different from 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 is not provided by the installer, it has to be added to the package manually. 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. This information is collected by the *cloudlet client* that runs on the mobile device. 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.

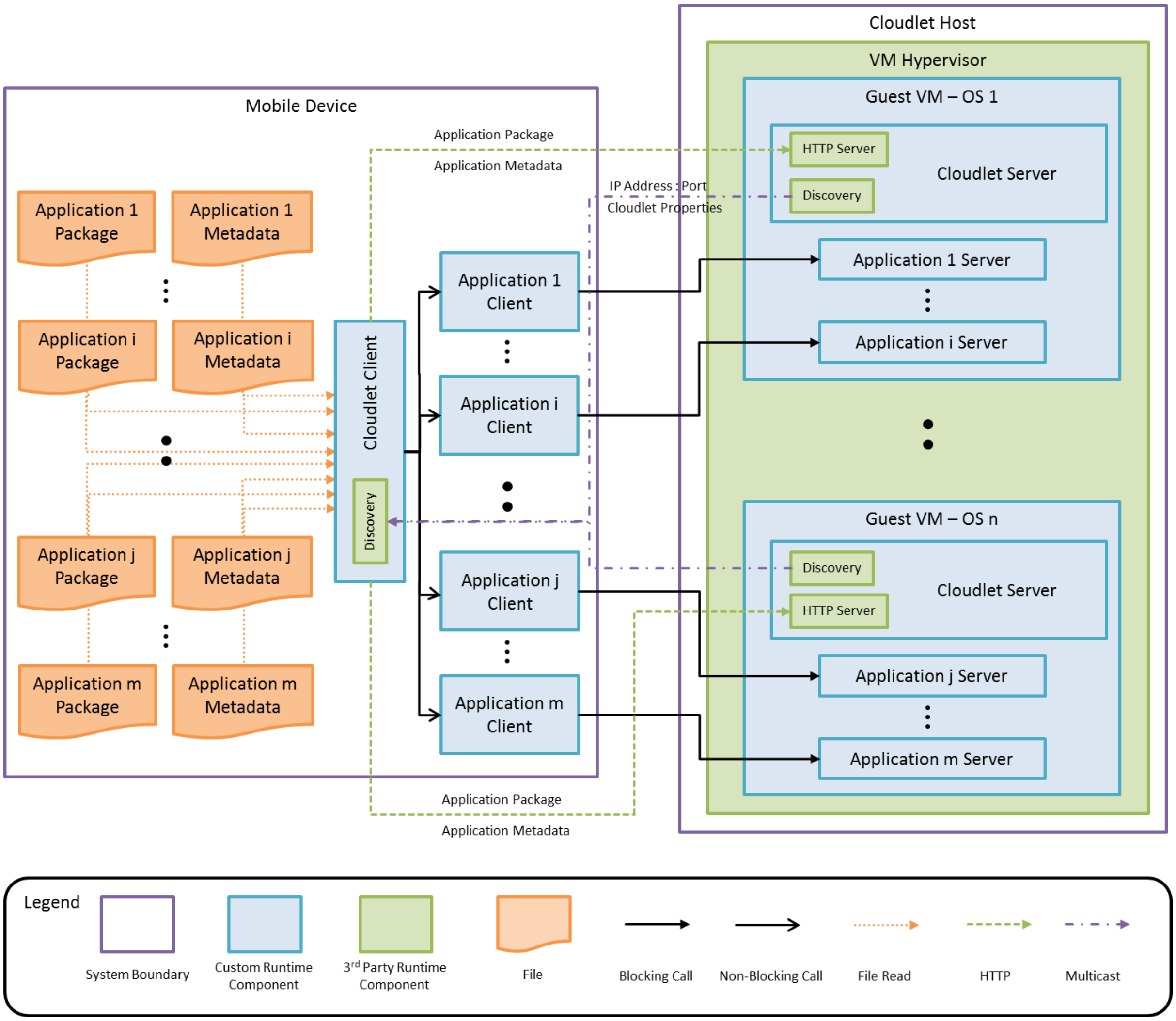

Figure 7: Application-Virtualization-Based Cloudlet Cyber-Foraging System Architecture

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 [29], which means that a VM has its own IP address.

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. The process of finding an appropriate cloudlet will be discussed 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 JRE 7 or higher. It contains a Jetty HTTP server [30] 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 [31], which is a pure Java implementation of multicast DNS and the zeroconf framework [32]. 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 are then carried out by the application server on its behalf. Afterwards, the computational result is sent back to the client. For example, the face recognition server that is mentioned in the cloudlet scenario in 3.3 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, the environment settings and other necessary files.

6.1.10 Application Metadata

Every application package is accompanied by a JSON [33] file that contains information about the package. The application metadata includes all cloudlet requirements that need to be met for a successful offload (cf. section 6.3.2).

6.2 Application Deployment Sequence

Figure 8 is a 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.

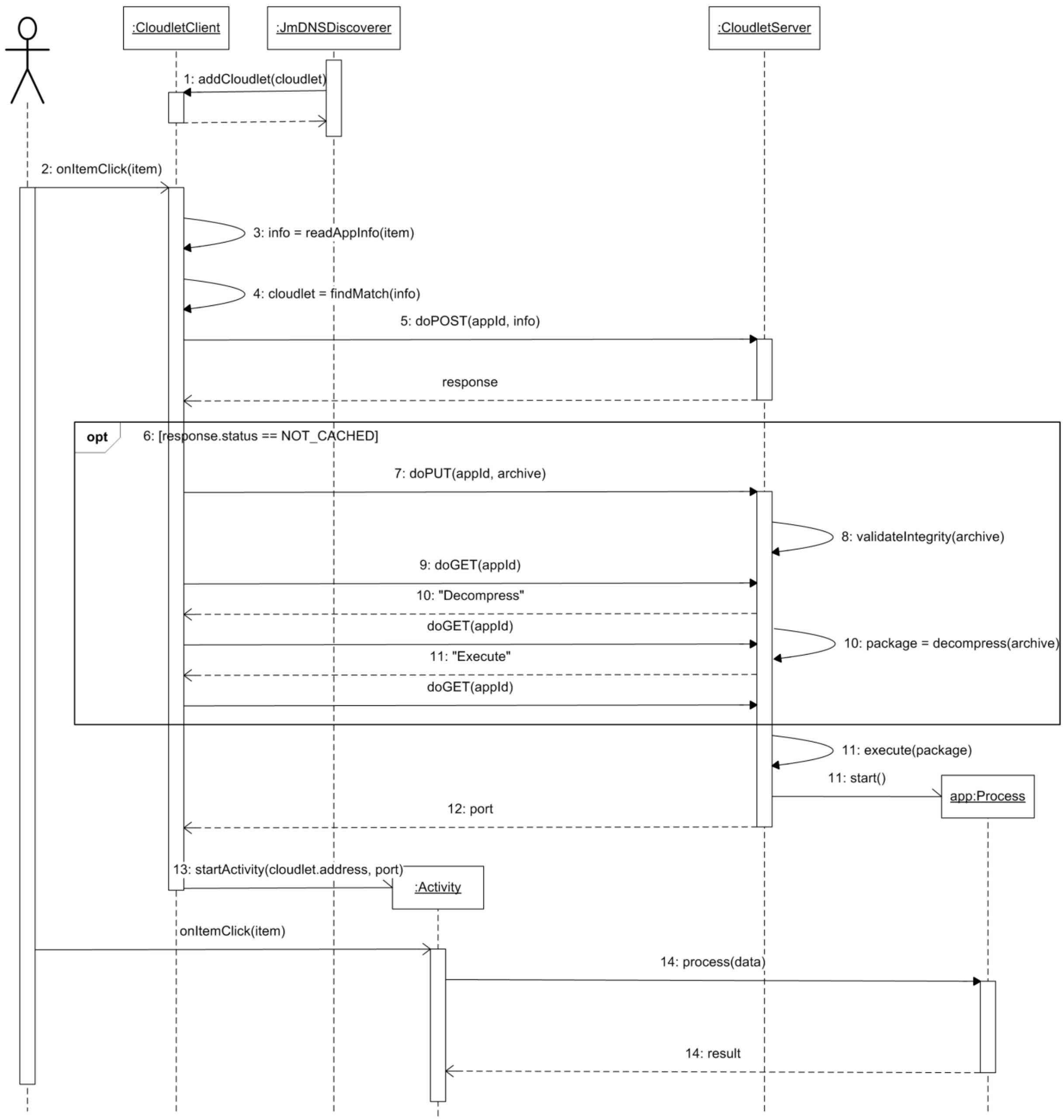

Figure 8: Application Deployment on a Cloudlet

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 integrity of the application package is validated 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 done in the context of this thesis. Selected design decisions are shown and discussed.

6.3.1 Cloudlet Server Code View

The cloudlet server code turns a VM into a code offload site. It 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 (cf. 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 knowledge of how to process uploaded applications is bundled inside the *packagehandler* package. This package contains interfaces and abstract classes that have to be implemented for each an application file type and required operating system. For this thesis, 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.

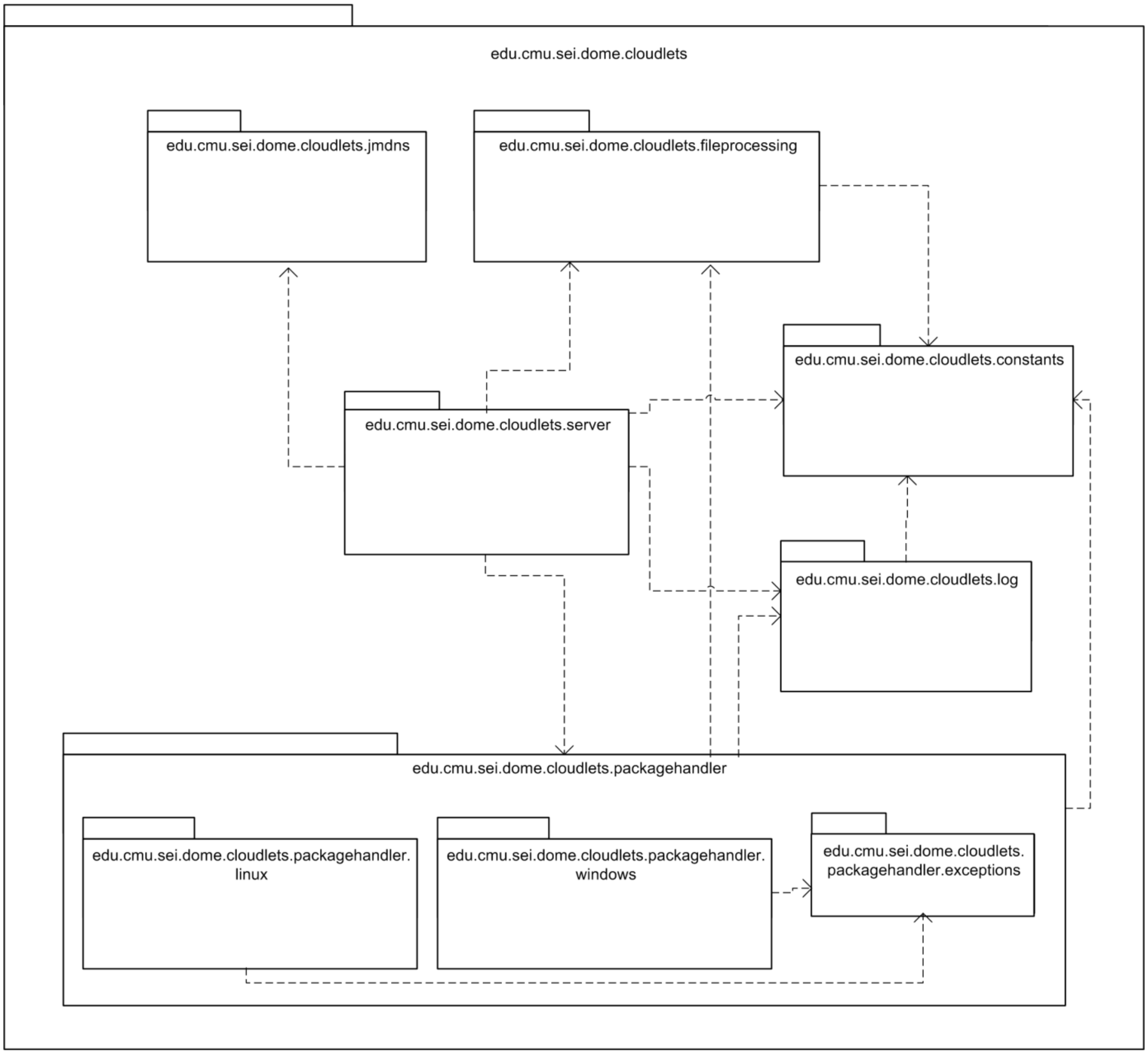

Figure 9: Cloudlet Server Package View

6.3.2 Application Metadata and Cloudlet Requirements Matching

The application metadata that accompanies each application package provides the information that is necessary for the cyber-foraging process. It is a JSON file whose structure is described by the schema in Figure 10. The *name* and *description* fields hold general information about the application; *checksum* and *size* are used by the cloudlet server 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 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.

```
{
    "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

While all other fields have basic types, *cloudlet* is an array of JSON objects. Each of these objects defines the set of requirements that need to 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

This principle is illustrated by the example in Figure 11. 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",
    "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* [34], is a web service design architecture that is centered on the concept of *resources*. 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.

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.

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

6.3.4 Long Polling

Using HTTP between a client and a server enforces a strict response-request scheme where all communication is initiated by the client only. There is no persistent connection between client and server; once the request has been answered, the HTTP connection terminates. As a consequence, the server has no other means to message the client than in the context of responding to a client request. However, in some cases it is useful to let the server push messages immediately to the client instead of waiting for the client to pull information through HTTP requests. In our cloudlet implementation, the server informs the client about the application deployment progress. Because the status on the server side determines the actual deployment status, the server should be able to notify the client as soon as the status changes.

One technique to allow the server initiating messages to the client is *long polling*. Long polling emulates a server push mechanism: a HTTP request is not served immediately but "conserved" by the server. When the server needs to send a message to the client, it responds

[1] There is research that shows that there may be collisions between md5 hashes [43], which is why they cannot guarantee uniqueness. A more advanced cryptographic hash function could minimize this risk.

to this conserved request. Therefore it autonomously determines the time to contact the client. As soon as the client receives the response it immediately initiates a new request that is then held again by the server until it decides to respond.

The cloudlet client starts to activate long polling directly after transmitting the application package to the cloudlet server. The cloudlet server then sends progress status notifications to the cloudlet, indicating what server action, such as decompression, is to be performed next. The user of the mobile device can therefore track the application deployment. He knows that the server is processing, while no information could otherwise confuse the user into thinking that the application transmission did not succeed.

The Jetty HTTP framework supports long polling through the concept of so-called *Continuations*. A continuation encapsulates a HTTP request and suspends it. When a response is to be sent, the continuation is completed; ergo the client request is answered.

In our implementation the long polling client is the *EventListener* class, which is a subclass to *Thread*. A response's HTTP status code determines how the response is handled; a status unequal to 400 and 410 causes the EventListener to issue a new GET request. Listing 1 presents 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 + "!");
        }
    }
}
```

Listing 1: Client Long Polling - EventListener.java

The server counterpart are the *RESTservlet* and *PushHandler* classes, which encapsulate the request and suspend or continue it, respectively. The main functionality is presented in Listing 2 and Listing 3.

```
@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);
}
```

Listing 2: Server Long Polling: RESTservlet.java

```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();
}
```

Listing 3: Server Long Polling - PushHandler.java

6.3.5 Bridge Pattern for OS Decoupling

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

In our implementation this is accomplished through the Bridge [35] design pattern. The Bridge pattern decouples abstraction from implementation; it thus facilitates changing the implementation without having to change the code that binds to the abstraction.

The *PackageHandler* class serves as the abstraction part in 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*'s *decompress* and *execute* methods get delegated to the concrete *PackageHandlerImpl*. Although not implemented, the Bridge pattern allows for an abstraction hierarchy that is independent from the hierarchy on the implementation side. For example, a *PackageHandler* subclass could examine if the application package complies with security demands before calling the *execute* function.

Each operating system family that is to be supported by the cloudlet server needs to implement the *PackageHandlerImpl* interface and provide the OS and application specific code. The cloudlet server for this thesis includes the *LinuxPackageHandler* and *WindowsPackageHandler*, both spanning separate class hierarchies which contain application specific classes. These application specific classes transitively inherit from the abstract

Executor class that is responsible for starting an application with submitted arguments. For example, the *LinuxPackageHandler* uses the *CDEExecutor* and *JARExecutor* classes, which encapsulate the knowledge how to handle CDE or JAR packages, respectively. Additionally, their direct superclass, the *LinuxTerminalExecutor*, embeds the application execution into a system terminal. Figure 12 visualizes the entire *PackageHandler* Bridge as a UML class diagram.

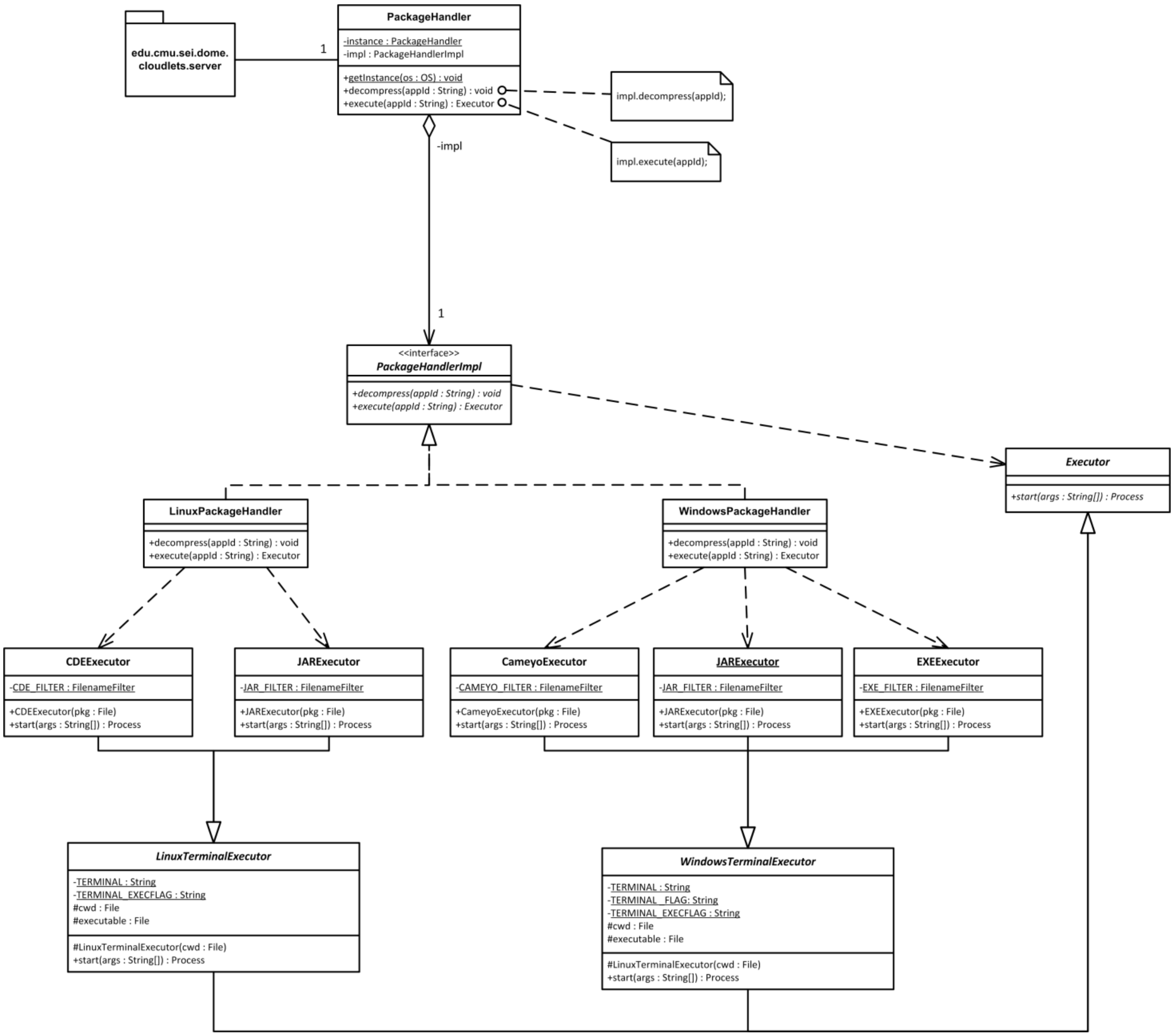

Figure 12: The Bridge Pattern decouples PackageHandler Abstraction from OS specific Implementation

7 Evaluation and Comparison with VM Synthesis

7.1 Functional Requirements

Referring to the functional requirements listed in 3.5.1, Application Virtualization and VM Synthesis are both legitimate strategies for cyber foraging which meet the listed requirements. They both serve the purpose of cloudlet use in hostile environments. This includes the principles of stateless servers that cannot rely on Internet access, thus both cloudlet implementations receive the actual application from the mobile client. In both cases, the deployment phase is preceded by a cloudlet discovery phase in which the mobile device finds suitable cloudlets by parsing the service information published by the cloudlets. Application Virtualization as well as VM Synthesis can guarantee the correct execution of a transmitted application if the transfer package or overlay has been created correctly. This creation process differs though in terms of simplicity and feasibility depending on the application that is to be made portable. Both strategies are able to return to a state with no traces of an offloaded application, i.e. a complete removal. Finally, they are capable of serving multiple clients simultaneously.

7.2 Quantitative Analysis

To identify the battery efficiency and speed of the application virtualization based cloudlet implementation it was evaluated using the following applications.

Object Recognition

The application server is a Linux C++ software 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. CDE was used to virtualize the application server.

Speech Recognition

Based on SPHINX, 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, 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 that matches its internal database.

NULL

Virtualized with CDE for Linux and with Cameyo for Windows respectively, the NULL application server is a C program which 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

<table>
<tr><th></th><th colspan="2">Object Recognition</th><th colspan="2">Speech Recognition</th><th colspan="2">Face Recognition</th><th colspan="2">NULL</th></tr>
<tr><td>Application size (MB)</td><td colspan="2">25.340</td><td colspan="2">100.140</td><td colspan="2">34,449</td><td colspan="2">0.009</td></tr>
<tr><td>Compressed Package size (MB)
(cde | cameyo)</td><td>28.492</td><td>-</td><td>67.748</td><td>65,370</td><td>-</td><td>13.090</td><td>1.133</td><td>0.940</td></tr>
</table>

7.2.1 Experiments

The experiments were conducted 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 (cf. Figure 13). 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 identifying the mobile device's energy consumption we used the Power Tool device and corresponding software from Monsoon Solutions [36].

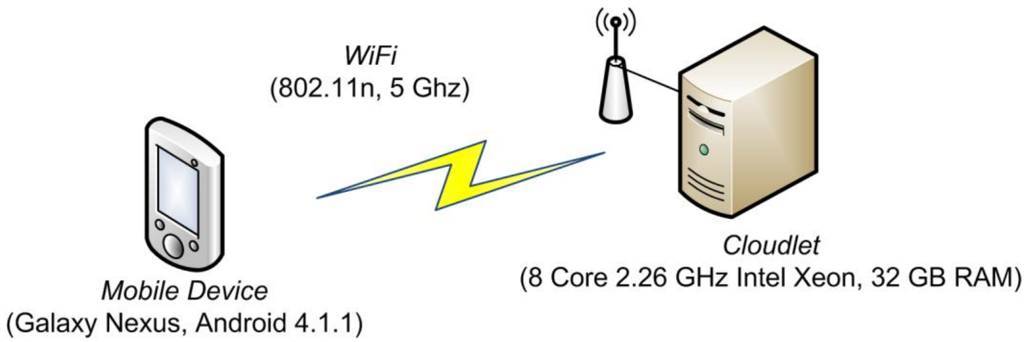

Figure 13: Evaluation Experimental Setup

Table 3 and Figure 14 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

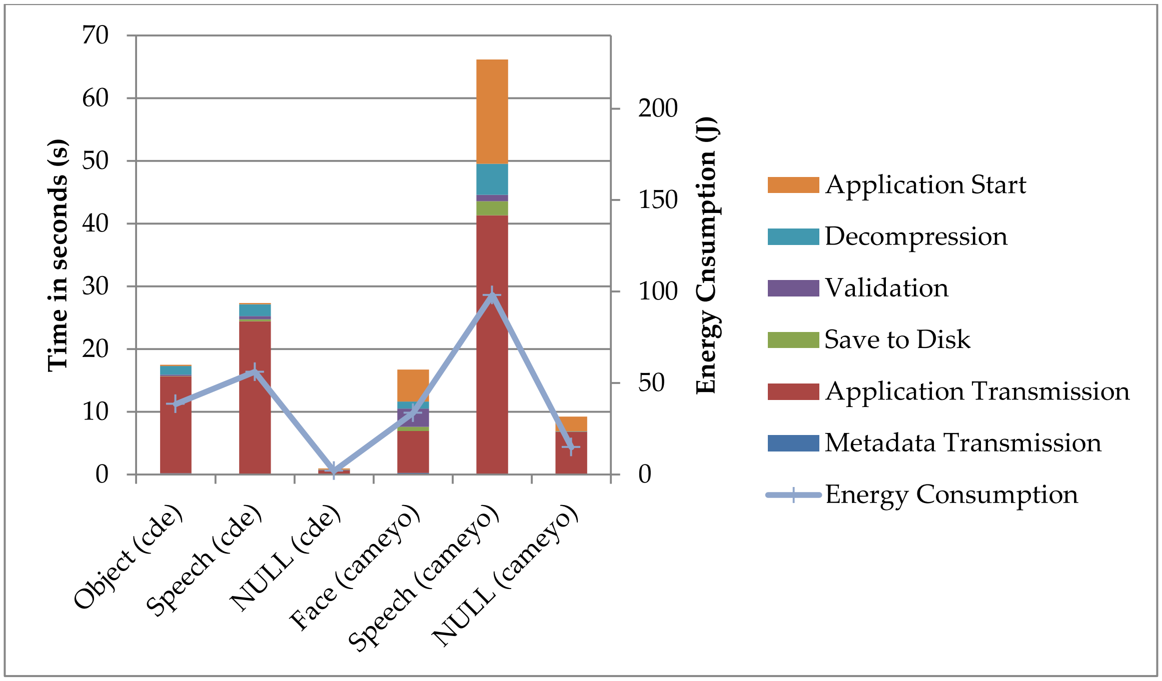

Figure 14: Time and Energy Measurements per Virtual Application

The deployment time ranged from 0.961 seconds (*NULL (cde)*) to 66.170 seconds (*Speech (cameyo)*). The lowest energy consumption could also be achieved by *NULL (cde)*, which consumed about 2 Joules, and the highest amount of energy was consumed by *Speech (cameyo)* with about 98 Joules. Application transmission time highly dominates the total 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 Conclusion

In the following, we divide the applications into two groups and observe their corresponding measurements separately. The reason for this is the experienced difference in performance that results from the employed virtualization tool and operating system. CDE applications which run on an Ubuntu 10.04 VM form the first group; Cameyo applications for Windows XP form the latter.

Setting application package size and deployment time into relation, there is a strong positive correlation for both CDE/Ubuntu10.04 and Cameyo/WinXP applications. The sample correlation coefficients are

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

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

Therefore, we suppose a linear dependence between package size and deployment time. An explanation for this observation is the linear dependence between file size and file transmission time, which is mostly determined by the Wi-Fi bandwidth. Another reason is the proportional relation between file size and the time needed for checksum computation and decompression. It also appears that Cameyo's application start time is proportional to file size (cf. Table 2 and Figure 14).

Linear regression indicates the following relations:

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

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

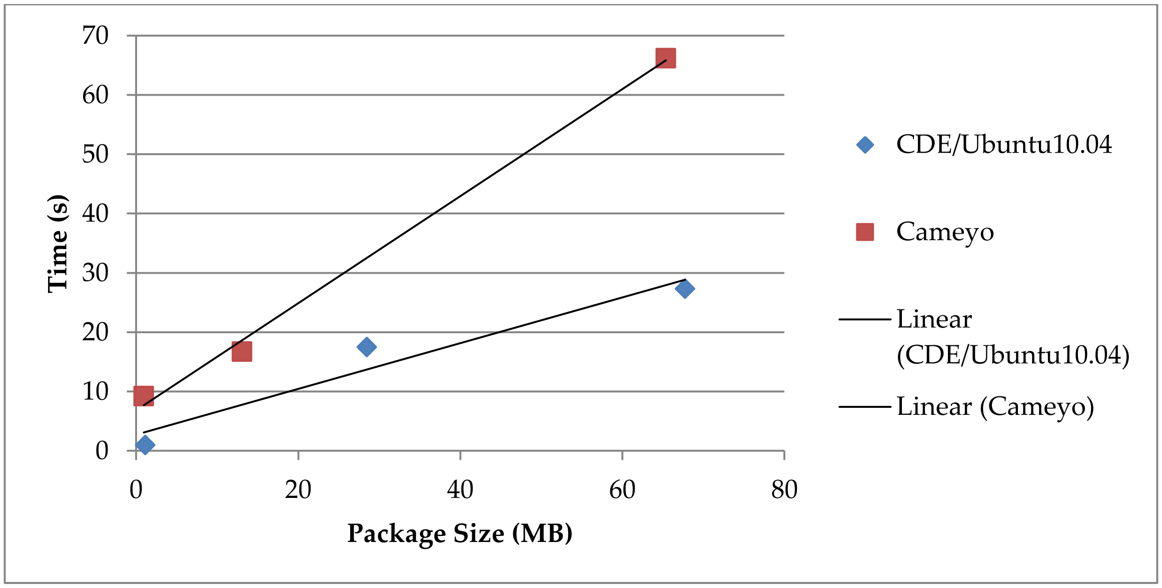

Figure 15: Application Package Size in relation to Deployment Time

During the experiments, a significant higher power level was measured on the mobile device during application transmission. Because Wi-Fi communication caused the major portion of total energy consumption and application transmission time is in linear relation to package size, there is a linear dependence between package size and energy consumption. This linear dependence can be observed in Figure 16.

The sample correlation coefficients are:

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

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

Linear regression:

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

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

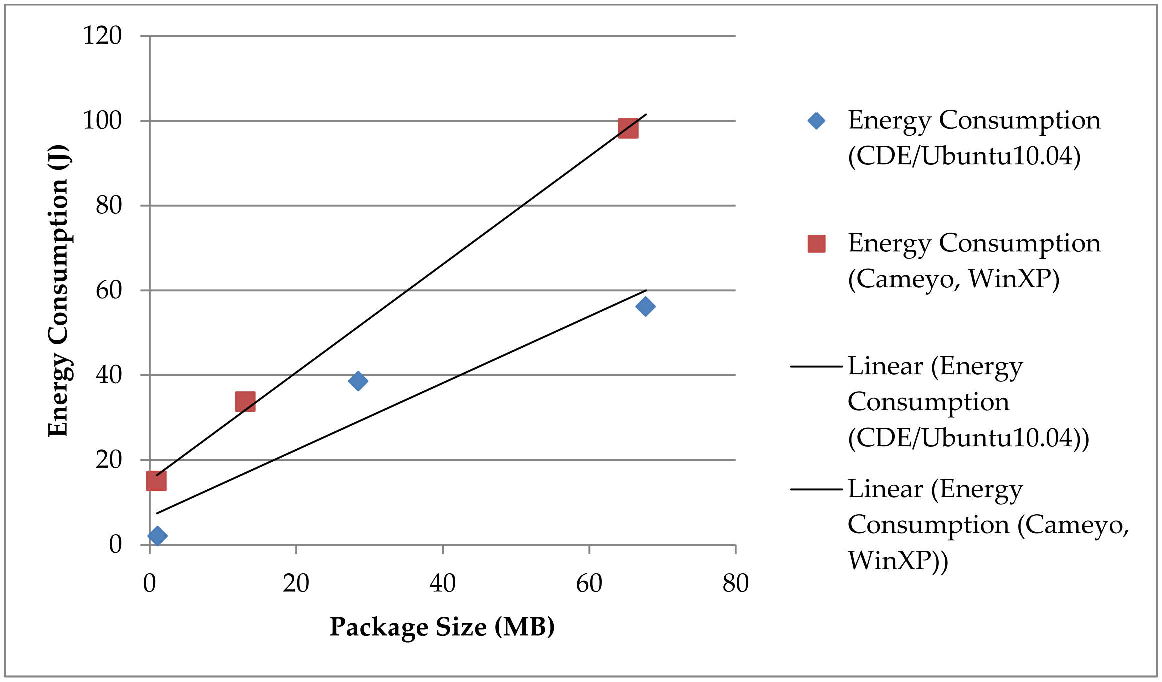

Figure 16: Application Package Size in relation to Energy Consumption

7.2.3 Comparison with VM Synthesis

The VM synthesis cloudlet reference architecture has been evaluated in [9]. Like in the application virtualization cloudlet implementation, the energy consumption and deployment time, or so-called "application ready time", increases with the amount of data that is to be transferred to the cloudlet machine. Considering the measurements from the revised VM synthesis prototype evaluation (p. 18ff) [9], a linear regression analysis shows a relation of $energy_J \approx 0.82 \times file\ size_{MB} + 7.93$ and $time_J \approx 0.68 \times file\ size_{MB} + 6.31$. Whereby "$file\ size_{MB}$" is the total amount of data, i.e. the sum of disk image overlay size and memory overlay size.

As a consequence of the experiments and the VM synthesis evaluation, a smaller package for transfer can drastically decrease the deployment time and energy consumption for cloudlet-based cyber foraging. Using the software from the VM synthesis implementation, disk and memory overlays of the same applications that were used for evaluating the application virtualization implementation were created. Hereby, 64-bit versions of Ubuntu 12.04 and Windows XP with Service Pack 2 were used. It turns out that the virtualized applications are significantly smaller in size than the combined overlays, cf. Table 4. Memory overlays tend to include data that is irrelevant to the application of interest, thus leading to an increased file size. Therefore, the application virtualization strategy outperforms VM synthesis in terms of fast deployment and low energy consumption.

Table 4: File Sizes of Application Virtualization vs. 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

7.3 Qualitative Analysis

In the last section, the performance of the application virtualization cloudlet solution has been evaluated. Therein, time and battery consumption during the application deployment phase were compared with the VM synthesis solution. This section discusses qualitative aspects of both solutions and emphasizes their differences in respect of consequences to using one strategy or the other.

7.3.1 Coupling between Application and Cloudlet

For the provision of a general cloudlet infrastructure that is capable to serve a large variety of different applications, the coupling between the application and its respective cloudlet is significant. A loose coupling facilitates the setup of cloudlet hosts that can act as surrogates to a large number of offload ready applications. A tight coupling instead urges the cloudlet host to provide more special environments that fit the particular application. *Application Virtualization* aims to separate applications from the underlying operating system. It achieves its goal insofar as it offers portability across distribution boundaries; e.g. a CDE application runs on various Linux distributions without need to adapt the virtualized application. Such an application cannot, however, cross operating system family boundaries; e.g. 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*, on the contrary, requires a target system on the cloudlet that is binary equal to the source system on which the application has been made ready for offload. The part that is being transferred to the cloudlet is the binary difference between two VM snapshots. Hence, in order to restore the final VM image, the cloudlet host needs to own the first snapshot, i.e. the base VM image, in advance.

As a consequence to the tight coupling between application and cloudlet when using VM synthesis, the mobile device requires the cloudlet host to have the correct base image. If this is not the case, the mobile client's overlay cannot be applied. A possible workaround is to let the mobile device transmit the complete final VM image. But this would lead to heavy costs regarding memory storage, deployment time and battery consumption because of the image's large file size.

However, if the mobile device transfers entire ready-to-use VM images, VM synthesis can provide a looser coupling than application virtualization because the cloudlet only needs to run a hypervisor that can handle the transferred images.

7.3.2 Patchability of the Target System

In the context of VM synthesis, base VMs cannot be updated without invalidating overlays that refer to it. To provide a secure and stable system, regular updates are necessary, but updates require a recomputation of overlays. The number of base VMs increases over time. As this number grows, the cloudlet infrastructure will get more scattered. This can lead to outdated base VMs that have to be kept for legacy reasons. Such would enforce an infrastructure where the mobile client relies on the cloudlet to provide a rare base VM. 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. Such is indispensable because a virtualized application is only as portable as the underlying runtime environment.

7.3.3 Range of Offload Ready Applications

Like the authors of CDE explain, applications "that require specialized hardware or device drivers" (p.7) [26] cannot be made portable across machines that do not meet these requirements. Nevertheless, Cameyo offers to package device drivers and temporarily integrate them into the operating system [37]. This approach only works for drivers that do not address the application's files and registry which are hidden within a sandbox. The device driver itself is not virtualized and runs outside the sandbox. This implies that it is not portable across system boundaries; the main goal is rather to avoid a driver installation on the target system.

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. But applications that use VM synthesis as its distribution mechanism also expect the cloudlet to have specialized hardware if it is required.

7.3.4 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. Like mentioned in the previous section, special hardware requirements may have to be provided by the cloudlet. These requirements have to be documented and then need to be negotiated with potential cloudlets.

Virtualized applications behave correctly as long as all of its 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, e.g. runtime environments like the JRE or device drivers. If one of these components appears to be incompatible, however, the execution will fail.

7.3.5 Application Preparation Overhead

Preparing an application for deployment on a cloudlet should require only low effort. VM synthesis and application virtualization do both require neither source code modifications nor insight into the application's source code. Both do thus not necessarily involve the application's developer himself to perform the offload preparation.

VM synthesis requires the initial installation on a base VM and then computes the overlay from the suspended state of the complete VM and the clean base VM (cf. Figure 17). This is a convenient mechanism and the main difficulty lies in the regular installation.

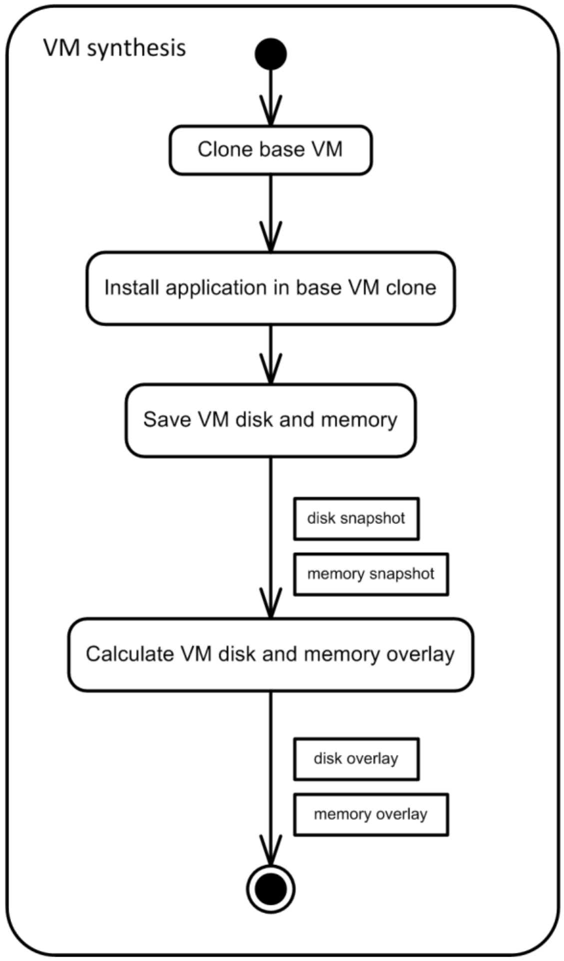

Figure 17: VM overlays Creation Process

Application virtualization can be accomplished in various ways. 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 offer modification of a created package, so instead of using one of the described mechanisms we can create an empty package and add files directly. Such a package modification is often necessary because the original supervision routine cannot guarantee to find all dependencies, cf. section 5.2. Therefore, deeper knowledge of the application's dependencies is necessary. Figure 18 shows the alternatives for creating a virtual application.

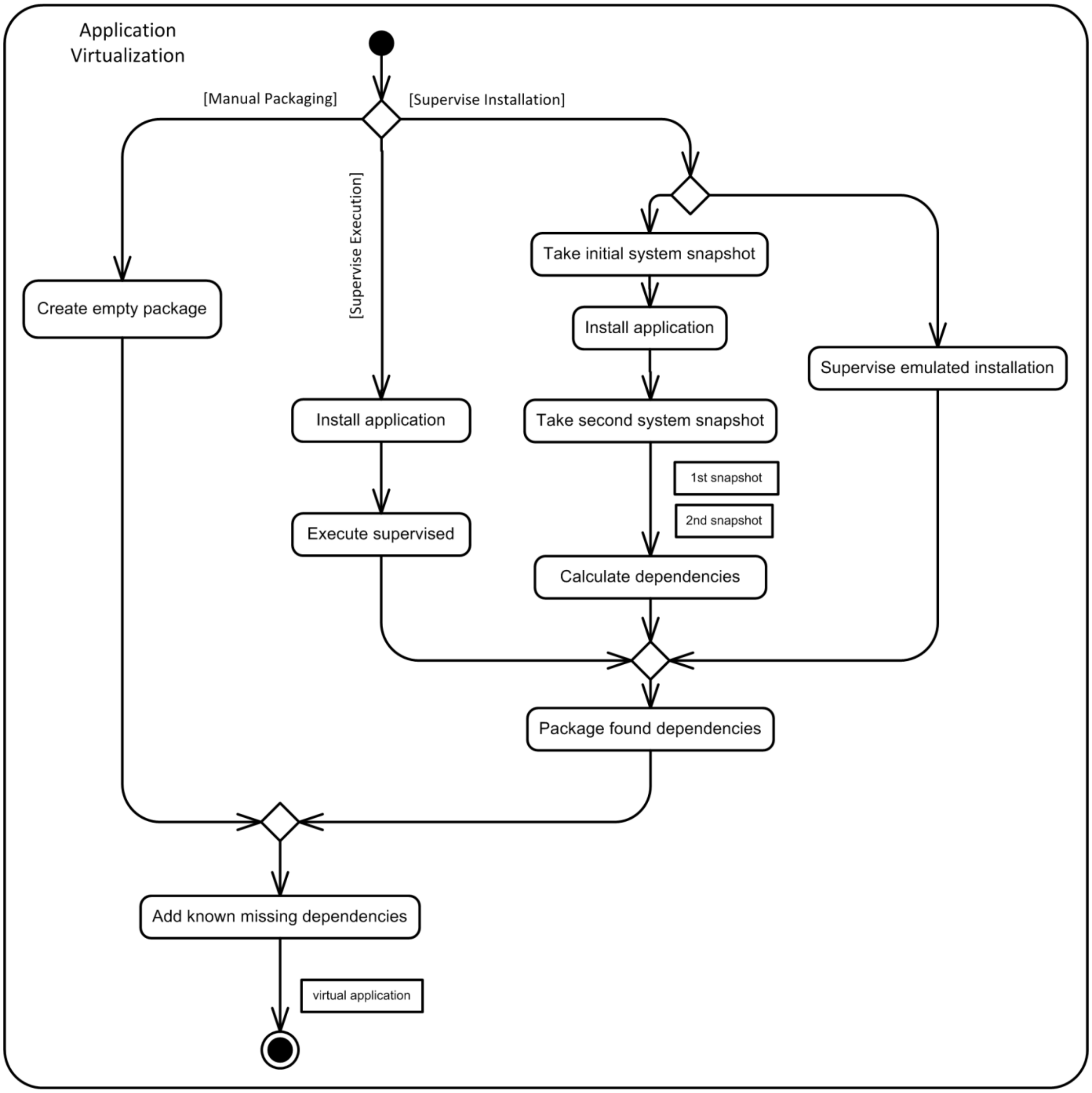

Figure 18: Virtual Application Creation Process Alternatives

7.3.6 Operation Overhead

For running an application server in the cloudlet implementation that was introduced in section 6, the server is embedded into an application virtualization runtime environment which in itself runs on a VM. These layers are shown in Figure 19. 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. The first switch occurs with the first system call, the kernel responds by causing a switch to the virtualization runtime which in return is followed by the final switch to the kernel for executing the modified system call. The authors of CDE measured the run-time slowdowns for their virtualized applications and found a slowdown rate ranging from 0% to 28%. Therein, due to system call frequency, CPU-bound applications had the smallest slowdowns and I/O-intensive tasks had the largest slowdowns (p. 13) [26].

The application virtualization runtime environment influences the execution performance, and so does the hardware virtualization layer. Hardware virtualization enables to execute the virtualized application on a VM rather than on the native OS. Hosting a guest OS within a VM induces both CPU overhead as well as memory overhead compared to running the OS directly on the physical hardware. The KVM virtualization performance has been evaluated by Larabel [38] on an Intel Core i7 machine. Accordingly, there is only a slight overhead for many computational tasks but a more significant one for disk-related tasks.

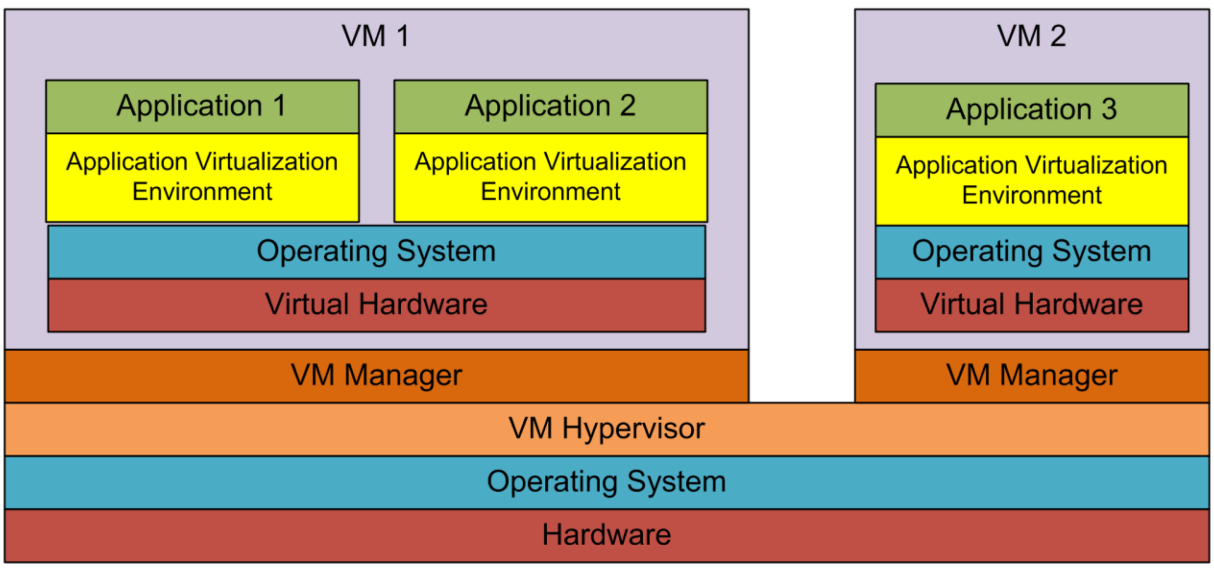

Figure 19: Application Virtualization Layer Architecture

The VM synthesis based implementation suffers from that same overhead that is imposed by running applications on a virtual rather than a physical machine. But it does not experience the overhead that is additionally caused by runtime environments like CDE or Cameyo. It utilizes, however, more VMs than the application virtualization based solution. Consider m Linux applications and n Windows applications: the VM synthesis implementation would have to manage $m + n$ separate VMs, while the application virtualization implementation would have to manage one Linux and one Windows VM with each of them running the appropriate applications in an application virtualization environment (cf. Figure 19 and in comparison Figure 20). VM synthesis will have higher costs on the host's resources than application virtualization when the number of applications that are run simultaneously increases. The workload of each VM includes the entire memory of the guest OS. Therefore, hosting multiple VMs, whose sum of guest OS memory is larger than the available memory on the host, forces the VM hypervisor to switch between VMs to offer application multitasking. Rather than scheduling at process level like is done in the application virtualization solution, the scheduling has to happen at VM level. Because a VM's workload is larger than the actual application's workload, a context switch is costly. Consider, for example, VMs where each uses 4 GB of memory and a cloudlet host with 32 GB of memory, which corresponds to the test scenario in section 7.2. Running many applications simultaneously soon exceeds the host's memory because every application runs on a separate VM. Swapping a VM that uses all of its available 4 GB of memory induces high costs in terms of disk operations, thus slowing down the overall cloudlet performance. Furthermore, it can be expected that real-world cloudlet applications such as face recognition have only limited expectations on system services. Providing a complete OS such as Ubuntu for only one single application includes system functionality that is not required for the specific application and adds to the computational overhead.

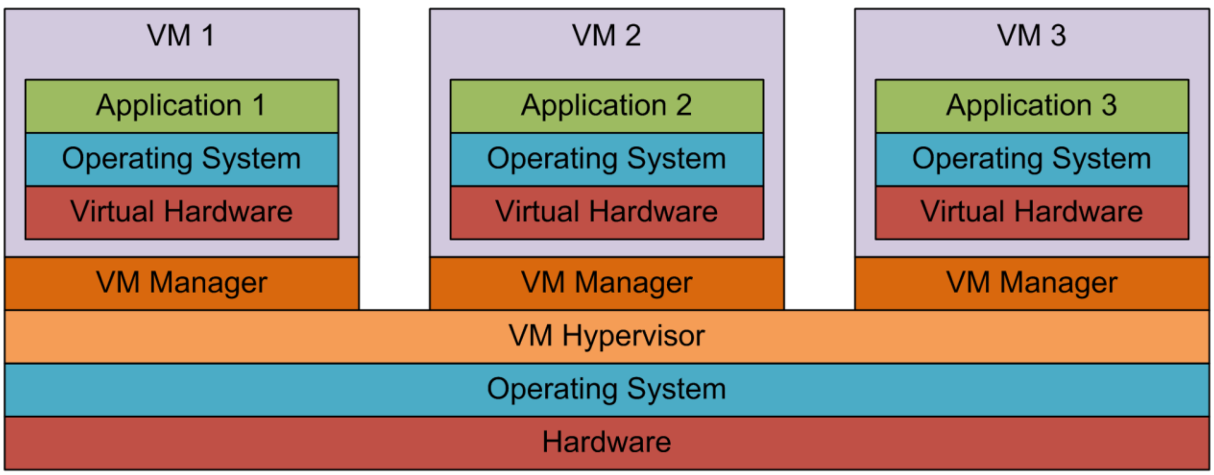

Figure 20: VM Synthesis Layer Architecture

7.3.7 Isolation and Security

Hardware virtualization adds a separate layer between the physical hardware and a guest OS. This layer is the VM hypervisor which either runs on the host operating system – a so-called *type 2* hypervisor - or directly on top of the physical hardware as a so-called *type 1* hypervisor. In both cases the VM on which the guest OS is running is isolated from an existing OS that runs natively on the physical hardware. Therefore, if the guest OS is compromised or malfunctioning, 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 implemented for this thesis as well as the VM synthesis implementation utilize this isolation benefit by running on VMs rather than on the native OS.

Another concern that is raised when discussing security is the isolation between different 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 to carry out a denial of service attack, thus blocking the physical hardware, or if it intrudes the commonly used network, the other VMs will be also harmed (p.6) [39]. Therefore, it is the hypervisors task to shut down misbehaving VMs.

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 VM level. CDE runs the packaged application within a *chroot jail* [40], thus preventing it to access files outside its package. This sandbox is, however, vulnerable to attacks that break the isolation mechanism [41]. Consequently, VM synthesis offers better isolation between applications than application virtualization.

8 Related Work

The idea of leveraging external resources to augment the capabilities of resource-limited mobile devices, termed as cyber foraging [8], has been researched by many. 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 either run 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. Analyzing the impact on performance metrics, the optimal profile is chosen, which then determines when to offload code to the remote machine. Examples of such cyber-foraging systems are Spectra [13], Chroma [14] [15] [16], MAUI [18] and Scavenger [17].

CloneCloud [19] follows the same code partitioning principle but automatically partitions code at the thread level without need for manual code annotation. 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 [22] enable a mobile device to trigger remote download and installation of applications on an external VM. This approach is closely related to the work presented in this thesis. A main difference is that the work in this thesis uses cloudlets as offload sites, which do not rely on Internet access. In addition, the cloudlet is not altered via remote installations which 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 thesis has been described in [3] and [6]. 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 efficiently establish this VM, a strategy called VM synthesis is implemented [3] [6] [42]. 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 [6]. The work in this thesis also focuses on providing external resources to mobile devices in hostile environments. However, instead of using a VM synthesis strategy, it explores the applicability of application virtualization as a strategy for cyber foraging. An architecture for application virtualization has been implemented as part of this thesis and is compared to VM synthesis.

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 automatically detect 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 thesis allow the manual addition of dependencies in order to create a complete a virtual package. Future work in this area should focus on facilitating the process of creating complete packages. A possible approach is to explicitly declare 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 thesis, and recommendation for future work, include:

- The implementation architecture presented in this thesis 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 has to 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, such as is done in VM synthesis.
- A real-world cloudlet solution has to satisfy security requirements, which have not been discussed in this thesis. 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 has to 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, i.e. resuming the halted application on another cloudlet while

preserving computational state and minimizing downtimes. Migration of virtualized applications may be 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 needs further exploration.

10 Conclusions

Cyber foraging, i.e. 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 thesis, we have focused on cyber foraging in hostile environments where reliable networks are not guaranteed. Especially, a connection to a distant cloud or data center 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 has to deploy its application server on the cloudlet. Related work that uses VM synthesis [3] [6] [42] for cloudlet provisioning has been the driver for this thesis.

The first part of this thesis explains the concept of cyber foraging in general, and then cloudlet-based cyber foraging in hostile environments in particular. Different strategies for cyber foraging are discussed and requirements for cloudlet-based cyber foraging are presented. We have explored application virtualization as a strategy for cyber foraging. The second part of this thesis provides 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 has been implemented. This implementation is introduced by presenting an architectural overview and discussing selected implementation decisions. Subsequently, the implementation has been evaluated and the achieved results and characteristics have been compared with the VM synthesis strategy. The final part identifies limitations of this work and presents some ideas for future work.

Summarizing 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 has to 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 thesis and try to overcome some of its shortcomings.

11 References

[1] "2011 handset and smartphone sales statistics worldwide: the big picture," 10 02 2012. [Online]. Available: http://mobithinking.com/blog/2011-handset-and-smartphone-sales-big-picture. [Accessed 08 10 2012].

[2] "Smartphone Users Around the World - Statistics and Facts [Infographic]," 02 01 2012. [Online]. Available: http://www.go-gulf.com/blog/smartphone. [Accessed 08 10 2012].

[3] M. Satyanarayanan, P. Bahl, R. Cáceres and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE CS Pervasive Computing,* pp. 14-23, October 2009.

[4] "Siri," Apple, [Online]. Available: http://www.apple.com/iphone/features/siri.html. [Accessed 31 07 2012].

[5] "Google Goggles," Google, [Online]. Available: http://www.google.com/mobile/goggles/. [Accessed 31 07 2012].

[6] K. Ha, G. Lewis, S. Simanta and M. Satyanarayanan, *Cloud Offload in Hostile Environments*, Pittsburgh, PA, USA: Carnegie Mellon University, 2011.

[7] E. Morris, "A New Approach for Handheld Devices in the Military," 4 April 2011. [Online]. Available: http://blog.sei.cmu.edu/post.cfm/a-new-approach-for-handheld-devices-in-the-military (2011. [Accessed 7 July 2012].

[8] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications*, pp. 10-17, August 2001.

[9] S. Simanta, G. Lewis, E. Morris, K. Ha and M. Satyanarayanan, "Cloud Computing at the Tactical Edge (CMU/SEI-2012-TN-015)," Software Engineering Institute, Carnegie Mellon University, 2012. [Online]. Available: http://www.sei.cmu.edu/library/abstracts/reports/12tn015.cfm.

[10] R. Chinnici, J.-J. Moreau, A. Ryman and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0," World Wide Web Consortium (W3C), 26 6 2007. [Online]. Available: http://www.w3.org/TR/wsdl20/. [Accessed 6 8 2012].

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

[12] Google, "Google Maps API Web Services," Google Inc., [Online]. Available: https://developers.google.com/maps/documentation/webservices/. [Accessed 6 8 2012].

[13] J. Flinn, D. Narayanan and M. Satyanarayanan, "Self-Tuned Remote Execution for Pervasive Computing," in *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, 2001.

[14] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen and H.-I. Yang, "The Case for Cyber Foraging," in *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, 2002.

[15] R. Balan, M. Satyanarayanan, S. Park and T. Okoshi, "Tactics-Based Remote Execution for Mobile Computing," in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys '03)*, San Francisco, CA, USA, 2003.

[16] R. K. Balan, D. Gergle, M. Satyanarayanan and J. Herbsleb, "Simplifying Cyber Foraging for Mobile Devices," in *Proceedings of the 5th International Conference on Mobile Systems Applications (MobiSys '07)*, San Juan, Puerto Rico, 2007.

[17] M. D. Kristensen, *Empowering Mobile Devices Through Cyber Foraging,* Aarhus, Denmark: Aarhus University, Department of Computer Science, 2010.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*, San Francisco, CA, USA, 2010.

[19] B. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "CloneCloud: Elastic Execution between Mobile Device and Cloud," in *Proceedings of the sixth conference on Computer systems (EuroSys '11)*, Salzburg, Austra, 2011.

[20] G. C. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, USA, 1999, pp. 187-200.

[21] Microsoft, "Component Object Model (COM)," Microsoft Corporation, 1993. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms680573(v=vs.85). [Accessed 6 8

2012].

[22] S. Goyal and J. Carter, "A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices," in *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems (WMCSA 2004)*, Lake District National Park, UK, 2004.

[23] A. Wolbach, "Improving the Deployability of Diamond," September 2008. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/anon/2008/CMU-CS-08-158.pdf. [Accessed 14 July 2012].

[24] I. Jackson and C. Schwarz, "Debian Policy Manual: Chapter 7 - Declaring relationships between packages," 04 03 2012. [Online]. Available: http://www.debian.org/doc/debian-policy/ch-relationships.html. [Accessed 18 09 2012].

[25] "RPM Packager Documentation: Dependencies," 22 09 2005. [Online]. Available: http://www.rpm.org/wiki/PackagerDocs/Dependencies. [Accessed 18 09 2012].

[26] P. J. Guo and D. Engler, "CDE: Using System Call Interposition to Automatically Create Portable Software Packages," in *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, USA, 2011.

[27] "ptrace (2) - Linux man page," [Online]. Available: http://linux.die.net/man/2/ptrace. [Accessed 14 8 2012].

[28] Cameyo, "Cameyo User Guide," 6 2012. [Online]. Available: http://cameyo.com/doc/CameyoManual.pdf. [Accessed 15 8 2012].

[29] KVM, "Setting guest network," 20 09 2011. [Online]. Available: http://www.linux-kvm.org/page/Networking. [Accessed 07 11 2012].

[30] "Jetty," 2012. [Online]. Available: http://www.eclipse.org/jetty/. [Accessed 18 09 2012].

[31] A. van Hoff, R. Blair and P. Frisch, "JmDNS," 25 08 2011. [Online]. Available: http://jmdns.sourceforge.net/. [Accessed 18 09 2012].

[32] "Zero Configuration Networking (Zeroconf)," [Online]. Available: http://www.zeroconf.org/. [Accessed 18 09 2012].

[33] D. Crockford, "RFC 4627 (The application/json Media Type for JavaScript Object Notation (JSON))," Network Working Group , 07 2006. [Online]. Available: http://tools.ietf.org/html/rfc4627. [Accessed 07 11 2012].

[34] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Irvine, CA, USA, University of California, 2000.

[35] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Bridge," in *Design Patterns*, Addison-Wesley, 1995, pp. 151-161.

[36] Monsoon Solutions Inc., "Power Monitor," 2008. [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/ . [Accessed 13 09 2012].

[37] "Latest Cameyo 2," Cameyo Blog, 25 04 2012. [Online]. Available: http://cameyoco.blogspot.com/2012/04/latest-cameyo-2.html. [Accessed 06 09 2012].

[38] M. Larabel, "Intel Core i7 Virtualization Performance," 22 4 2009. [Online]. Available: http://www.phoronix.com/scan.php?page=article&item=intel_corei7_virt&num=1. [Accessed 9 9 2012].

[39] Q. Chen, R. Mehrotra, A. Dubeyy, S. Abdelwahed and K. Rowland, "On State of The Art in Virtual Machine Security," in *Southeastcon, 2012 Proceedings of IEEE*, Orlando, Florida, USA, 2012.

[40] FreeBSD System Calls Manual, "chroot (2) - FreeBSD System Calls Manual," 04 06 1993. [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2. [Accessed 10 09 2012].

[41] Simes, "How to break out of a chroot() jail," 12 05 2002. [Online]. Available: http://www.bpfh.net/simes/computing/chroot-break.html. [Accessed 10 09 2012].

[42] S. Simanta, G. Lewis, E. Morris, K. Ha and M. Satyanarayanan, *Cloud Computing at the Tactical Edge,* Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2012.

[43] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *Advances in Cryptology – EUROCRYPT 2005*, Aarhus, Denmark, 2005.