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



Fakultät für Informatik

|  |
| --- |
|  |

**Dominik Messinger**

September 23, 2012

Bachelor Thesis

School of Computer Science

**Supervisors:**

Prof. Dr. Ralf Reussner

Dr. Grace Lewis

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.

Pittsburgh, den 23. September 2012

*Dominik Messinger*

**Abstract**

While modern mobile devices create new opportunities to interact with the surrounding environment, their computational power and battery capacity is limited. Code offloading to external machines that are located in clouds or data centers can help to overcome these limitations and provide necessary resources for compute-intensive tasks such as speech recognition, image processing or decision-making. However, in hostile environments, such as theaters of military operations or natural disaster, 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 machines in single-hop wireless distance, so-called cloudlets. Cloudlets can substitute cloud access in the context of hostile environments. VM synthesis is a strategy based on virtual machines that enables mobile devices to deploy custom applications on a cloudlet. While its general applicability has been shown in related work, the deployment process is slow 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](#_Toc335918990)

[1.1 Background and Motivation 1](#_Toc335918991)

[1.1.1 WAN Latency as a Limitation to Cloud Resources 1](#_Toc335918992)

[1.1.2 Hostile Environments without WAN Access 2](#_Toc335918993)

[1.1.3 Cyber Foraging and VM Synthesis in Hostile Environments 2](#_Toc335918994)

[1.2 Goal and Structure of this Thesis 4](#_Toc335918995)

[2 Cyber Foraging 5](#_Toc335918996)

[2.1 Concept 5](#_Toc335918997)

[2.2 Scenario 5](#_Toc335918998)

[2.3 Cyber Foraging Strategies 6](#_Toc335918999)

[2.3.1 Pre-installed Applications 6](#_Toc335919000)

[2.3.2 Mobile Code 7](#_Toc335919001)

[2.3.3 Application Deployment 7](#_Toc335919002)

[2.3.4 Virtual Machine Deployment 8](#_Toc335919003)

[3 Cloudlets 9](#_Toc335919004)

[3.1 Concept 9](#_Toc335919005)

[3.2 Architecture 9](#_Toc335919006)

[3.3 Cloudlet Scenario 10](#_Toc335919007)

[3.4 Phases of Cloudlet Interaction 11](#_Toc335919008)

[3.5 Cloudlet Requirement Analysis 13](#_Toc335919009)

[3.5.1 Functional Requirements 13](#_Toc335919010)

[3.5.2 Nonfunctional Requirements 13](#_Toc335919011)

[4 Application Deployment 14](#_Toc335919012)

[4.1 Limitations to Portability 14](#_Toc335919013)

[4.1.1 Instruction Set Architecture 14](#_Toc335919014)

[4.1.2 Hardware Dependencies 14](#_Toc335919015)

[4.1.3 Software Dependencies 14](#_Toc335919016)

[4.1.4 Dependency Conflicts 15](#_Toc335919017)

[4.2 Source Code versus Binary File Transmission 16](#_Toc335919018)

[4.3 Packaging Dependencies 16](#_Toc335919019)

[4.3.1 Remote Install 16](#_Toc335919020)

[4.3.2 Library Packaging 16](#_Toc335919021)

[4.3.3 Static Linking 17](#_Toc335919022)

[4.3.4 Application Virtualization 17](#_Toc335919023)

[5 Application Virtualization for Cloudlets 19](#_Toc335919024)

[5.1 Design Goals 19](#_Toc335919025)

[5.2 Application Virtualization Tools 20](#_Toc335919026)

[5.2.1 CDE 20](#_Toc335919027)

[5.2.2 Cameyo 20](#_Toc335919028)

[6 Implementation 22](#_Toc335919029)

[6.1 Basic Architecture 22](#_Toc335919030)

[6.1.1 Mobile Device 23](#_Toc335919031)

[6.1.2 Cloudlet Host 23](#_Toc335919032)

[6.1.3 VM Hypervisor 23](#_Toc335919033)

[6.1.4 Cloudlet Client 23](#_Toc335919034)

[6.1.5 Cloudlet Server 24](#_Toc335919035)

[6.1.6 Discovery 24](#_Toc335919036)

[6.1.7 Application Client 24](#_Toc335919037)

[6.1.8 Application Server 24](#_Toc335919038)

[6.1.9 Application Package 25](#_Toc335919039)

[6.1.10 Application Metadata 25](#_Toc335919040)

[6.2 Application Deployment Sequence 25](#_Toc335919041)

[6.3 Details 27](#_Toc335919042)

[6.3.1 Cloudlet Server Package Architecture 27](#_Toc335919043)

[6.3.2 Application Metadata and Cloudlet Requirements Matching 28](#_Toc335919044)

[6.3.3 RESTful Architecture 31](#_Toc335919045)

[6.3.4 Long Polling 32](#_Toc335919046)

[6.3.5 Bridge Pattern for OS Decoupling 34](#_Toc335919047)

[7 Evaluation and Comparison with VM Synthesis 37](#_Toc335919048)

[7.1 Functional Requirements 37](#_Toc335919049)

[7.2 Quantitative Analysis 37](#_Toc335919050)

[7.2.1 Experiments 38](#_Toc335919051)

[7.2.2 Conclusion 40](#_Toc335919052)

[7.2.3 Comparison to VM Synthesis 42](#_Toc335919053)

[7.3 Qualitative Analysis 43](#_Toc335919054)

[7.3.1 Coupling between Application and Cloudlet 43](#_Toc335919055)

[7.3.2 Patchability of the Target System 44](#_Toc335919056)

[7.3.3 Range of Offload Ready Applications 44](#_Toc335919057)

[7.3.4 Correct Operation 45](#_Toc335919058)

[7.3.5 Application Preparation Overhead 45](#_Toc335919059)

[7.3.6 Operation Overhead 47](#_Toc335919060)

[7.3.7 Isolation and Security 49](#_Toc335919061)

[8 Related Work 51](#_Toc335919062)

[9 Limitations and Future Work 53](#_Toc335919063)

[10 Conclusion 55](#_Toc335919064)

[11 References 57](#_Toc335919065)

# Introduction

## 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, a whole ecosystem of applications evolved that today shape the way its users interact with the world surrounding them. Context-aware services such as localization help to find nearby venues or tell 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, which exceeds our natural recognition capabilities and thereby augments the human’s perception, is already 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 [1]. To encounter the issue of resource-poorness, techniques have been developed that offer the mobile device 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’s capabilities.

Popular commercial cloud-connected mobile applications are the natural language processing software Siri, which needs access to the Apple Cloud [2], or Google Goggles communicating with Google servers for providing image recognition [3]. Using cloud computing, 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.

### WAN Latency as a Limitation to Cloud Resources

One main drawback of relying on cloud computing 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. There is a physical lower bound to latency since we cannot surpass light speed. Additionally, latency increases through packet drops and various software or hardware layers, such as routing mechanisms, congestion avoidance, integrity checks or security layers. It has been assumed that WAN latency is not going to improve since modern networking research focuses on issues such as security and manageability [1]. Solutions to these issues often lead to an increase in 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, we measured that the average round-trip time between Pittsburgh and San Francisco is 81ms. Transmitting bitmaps of negligible file size (1 bit) and omitting processing times will result in a maximum frame rate of 12 FPS, which is insufficient for highly responsive augmented reality applications (c.f. [4]). As long as services are bound to predetermined servers, these have to be close enough to the mobile device in order to provide a satisfying performance.

### Hostile Environments without WAN Access

The assumption of good Internet access, which satisfies the mobile device’s demand on bandwidth and reliability, may be wrong in some cases. Cloud computing is infeasible in unreliable networks, which occur in theaters of military operations or in the context of disaster recovery. Such conditions have been described as hostile environments by Ha et al. [5]. Hostile environments, in contrast to areas with well-established network infrastructures, forbid trust in 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 [5]. The Department of Defense has shown a strong interest in equipping soldiers with handheld devices to enhance their operational abilities [6]. 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.

### 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 [5]. It utilizes cyber foraging techniques in single-hop networks and virtualization technology. The term cyber foraging – first introduced in [7] – 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) [5].

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 virtual machine images and receives VM overlays that enable the cloudlet to reconstruct a complete virtual machine 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. The overlay receiving cloudlet merges this delta and the base VM; this results in a complete system that is ready for code offloading. 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 [8]; Figure 1 shows the process of the overlay creation.

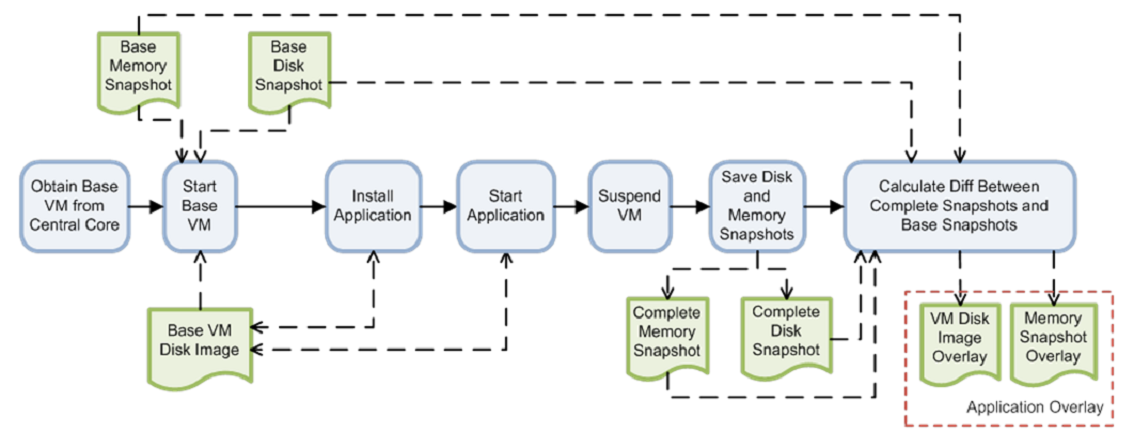


Figure 1: VM overlay creation (p.16) [8]

Cyber foraging in combination with VM synthesis offers an uncomplicated solution to encounter 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 the overlay transmission. Overlays tend to be significantly larger than the pure application server because the overlay creation process includes information that is irrelevant to the application. Transfer time between mobile and cloudlet, as well as battery consumption increases proportionally to the overlay size (cf. p. 19) [8]. Regarding flexibility, VM synthesis expects to match a VM overlay with its corresponding base VM during creation time. Therefore, updates to the base VM require recomputation of the overlay, because this is based on the binary difference.

## Goal and Structure of this Thesis

Subject of this thesis is to explore the applicability of application virtualization as a strategy for cyber foraging in hostile environments. Application virtualization emulates operating system services for applications; this approach is more lightweight compared to 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.

# Cyber Foraging

## 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 wired machines. The requirements for mobile devices - such as small 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 or decision making are amongst the most desired applications for mobile computing [1].

Cyber foraging, as first introduced by Satyanarayanan [7], 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 [7] machine, taking advantage of a more powerful hardware infrastructure. This surrogate executes the code and returns the computational result to its client.

## Scenario

Consider the following scenario for cyber foraging where the surrogate is part of a cloud.

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. 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.

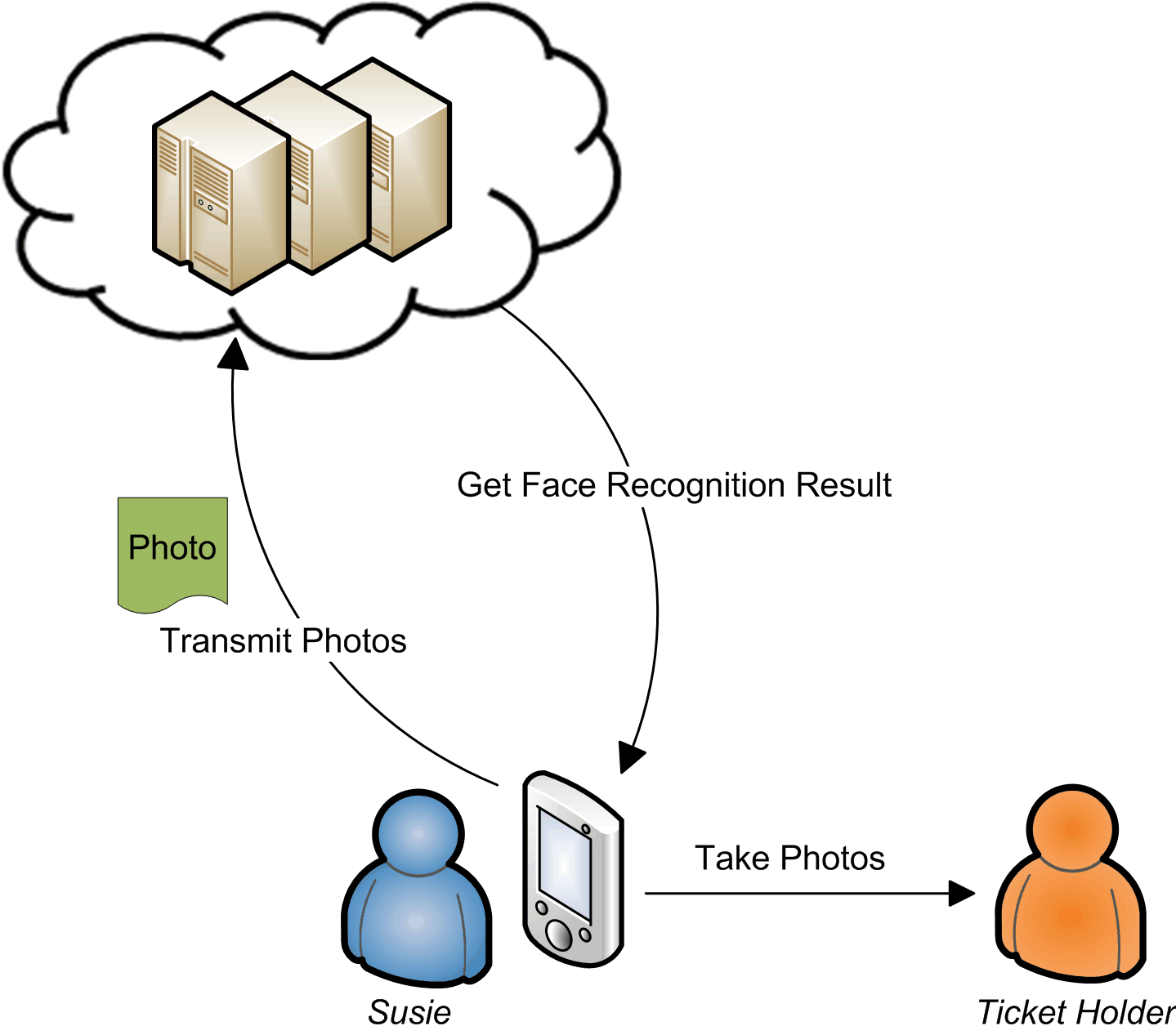


Figure 2: Cyber foraging scenario “Stadium Security”

## Cyber Foraging Strategies

In order to relieve the mobile client, the surrogate machine needs qualify for 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 gives some concrete examples of cyber foraging systems; they are categorized regarding the strategy they implement.

### Pre-installed Applications

If the cloudlet is ready for code offload, nothing further needs to be deployed. This is hence 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 [9], *CORBA IDL 3.5* [10]).

A real world cyber foraging example could be a smartphone app which uses the Google Maps API Web Services [11] for computing shortest distances.

### Mobile Code

A different deployment approach lets the surrogate execute portions of code that have been transferred from the client. This is different to 2.3.1 because not only data – i.e. service identifier plus arguments – but code is transmitted. Using web services or remote procedure calls, all code would already exist on the surrogate.

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

Several cyber foraging systems have been proposed that follow the mobile code approach. Spectra [12] and its successor Chroma [13] require the developer to modify the source to identify code for remote execution. He can influence how the remote execution is performed by setting quality demands and, in the case of Chroma, by defining tactics that declare alternatives for sequential or parallel operation of remote procedures [14] [15]. Similar solutions are Scavenger [16] and MAUI [17], which support code annotations to mark procedures for remote execution.

Designed to let mobile devices take benefit of a cloud, CloneCloud [18] 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 seamlessly migrates to the cloud and returns back to the mobile device. An older automatic partitioner that is unrelated to mobile computing is Coign [19]. Restricted to the Microsoft Component Object Model (COM) [20], Coign identifies components for remote placement by intercepting inter-component communication.

### Application Deployment

Another way of implementing cyber foraging is to deploy a self-contained application on the surrogate. After the installation finished, the mobile client can then communicate with the application and pass it resource-intensive work. Application Deployment facilitates the architecture since there is no complex middleware like 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.

Goyal and Carter [21] introduced a cyber foraging system that implements application deployment. The mobile device triggers the surrogate to download the requested application from the Internet and install it within its own virtual machine, where it is isolated from other applications.

### Virtual Machine Deployment

The artifact of virtual machine deployment is a complete virtual machine that contains the installed application. Different from application deployment, the deployment process is not the application being installed on the VM, but the entire VM being deployed on the surrogate.

Satyanarayanan et al. use a technique called VM synthesis to provision the surrogate machine [1]. The binary difference between VM snapshots that are taken before and after application installation is computed and send over to the surrogate machine. The surrogate, which owns the original base VM without the installation, can then restore the application-ready VM. This technique is first described in [22].

The work in this thesis uses application virtualization as a cyber foraging strategy and evaluates it in comparison with the VM synthesis strategy. Application virtualization belongs to the category of application deployment (2.3.3), VM synthesis to the category of virtual machine deployment (2.3.4).

# Cloudlets

## Concept

In spite of relying on distant computing clusters like clouds, Satyanarayanan et al. propose the usage of so-called cloudlets. A cloudlet is a computer or computer cluster that serves as a code offloading site for nearby mobile devices [1]. The close one-hop proximity to such a cyber foraging surrogate avoids possibly high latencies (c.f. [1]). Cloudlets gain advantage from the benefits of local area networks such as low-latency, high bandwidth and less vulnerability to cyber-attacks compared with wide area networks like the Internet (p.15) [1](p.4) [5]. In contrast to cloud computing, 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) [1].

## Architecture

Ha et al. propose a two-level hierarchy with cloudlets at the edge and a cloud at the core [5]. The cloudlets serve as offload elements for mobile devices; the cloud’s only task is to provision the cloudlets, which no longer have to access the cloud afterwards. 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. Since no essential state is kept, it takes little effort to install or replace a cloudlet. The motivation for this architecture has been the supply with computing power in hostile environments (p.8) [5]. In contrast, Satyanarayanan et al. assumed in [1] that cloudlets would have permanent Internet access (p.14).

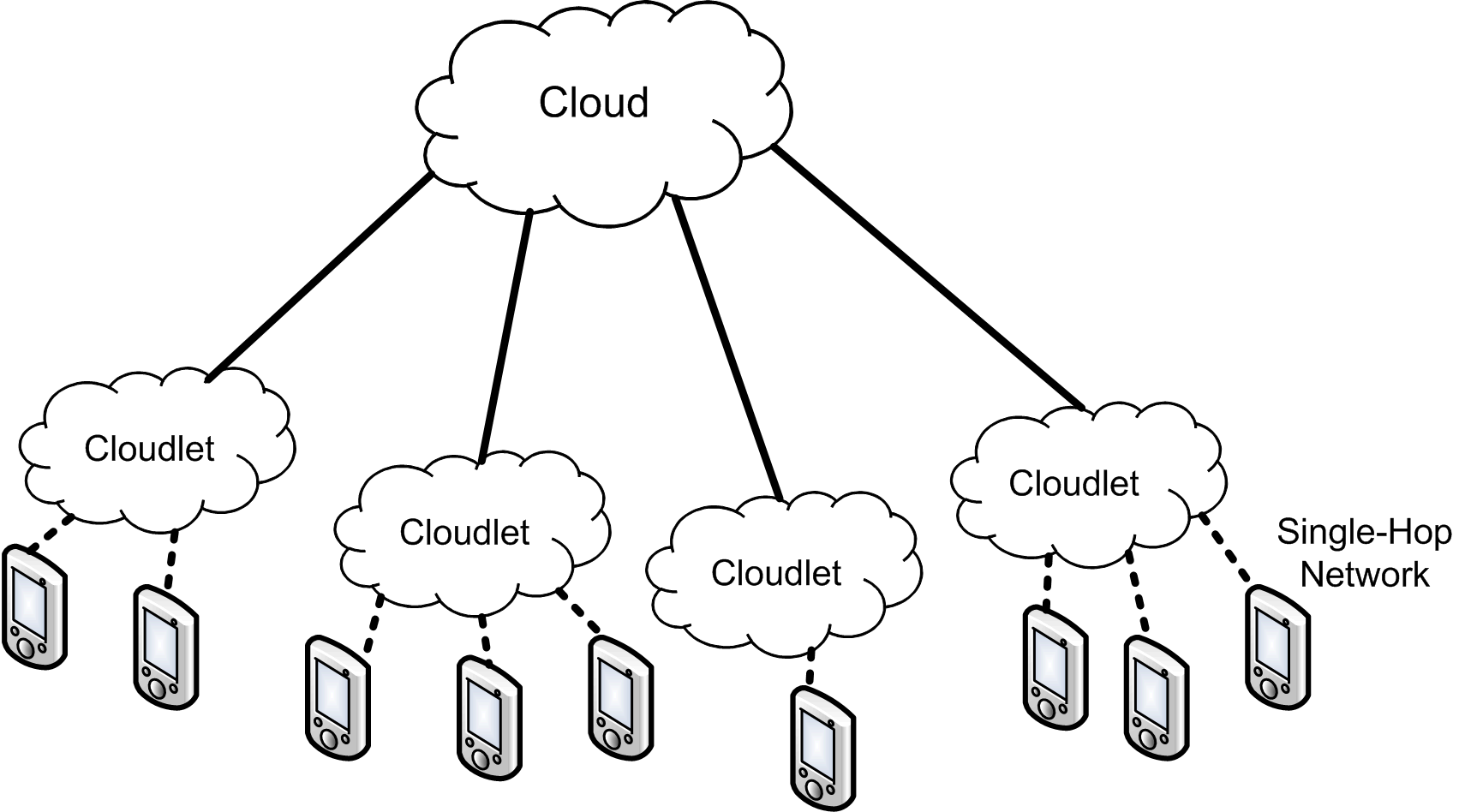


Figure 3: Cloudlet Architecture based on Ha et al. [5]

## Cloudlet Scenario

The following cyber foraging scenario is set in a hostile environment, which prohibits the reliance on the Internet.

Susie quit her job as a security guard and now works for a NGO that focuses on disaster recovery. Recently, a massive earthquake has occurred, which cost a high number of casualties and left a whole region in pure chaos. Many houses have been destroyed and the telecommunication infrastructure is out of order. Susie’s task is to go to countryside villages and interview the survivors; therefore gaining an oversight 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 the cloudlet can receive speech recordings from her phone; after successful speech recognition it then returns a translated text transcript. Finally, Susie’s phone uses text-to-speech functionality to voice her sentence translated to the local language. Vice versa, her phone enables her to understand the people that she interviews.



Figure 4: Cloudlet Scenario

## Phases of Cloudlet Interaction

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

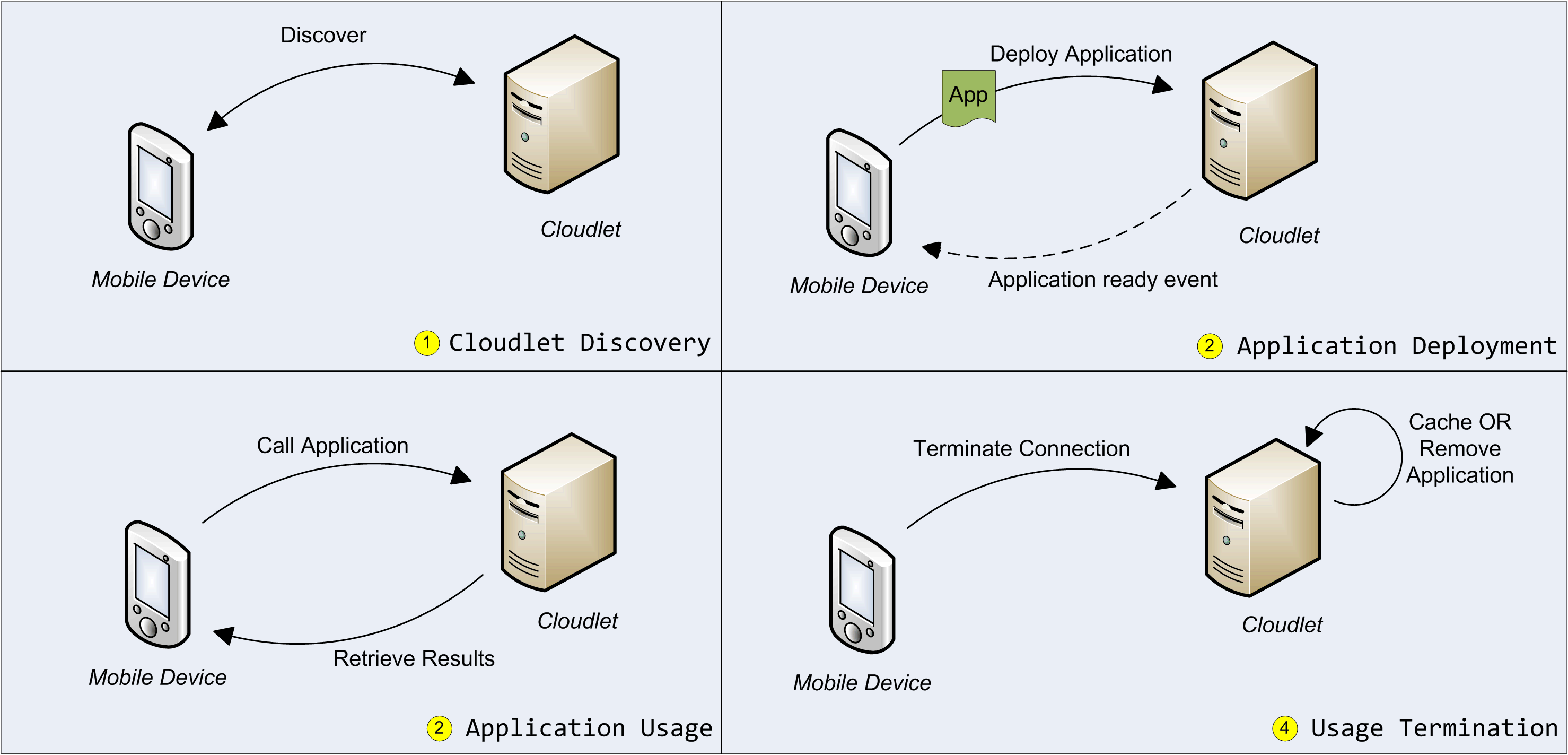


Figure 5: Phases of Cloudlet Interaction

1. Cloudlet Discovery

The mobile device first discovers nearby cloudlets and chooses which cloudlet meets its requirements best. Therefore, each cloudlet has to publish information about itself, which are retrieved through a discovery mechanism.

1. Application Deployment

After the mobile device has found 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 otherwise.

1. Application Usage

As soon as the application is deployed on the cloudlet, the mobile device starts to interact with it in order to accomplish the tasks that should be offloaded to the cloudlet.

1. 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.

## Cloudlet Requirement Analysis

Outgoing from the vision of easy to deploy and general multi-purpose cloudlets that should transiently be of service for mobile device users, the following functional and nonfunctional requirements can be elicited.

### Functional Requirements

1. A mobile device must discover all cloudlets, which are in the same wireless network.
2. Each cloudlet must publish information about its characteristics, thus allowing the mobile device to find the most suitable cloudlet for its application.
3. A mobile device must choose a suitable cloudlet for application deployment if available; otherwise it should give information about the unavailability.
4. Each application must hold information about its requirements on 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.

### Nonfunctional Requirements

1. The application deployment should be as fast as possible.
2. The interaction with the cloudlet should cost the mobile device as less energy as possible.
3. A cloudlet should be general, 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 fast and simple.
6. Making an application ready to use on a cloudlet should be simple.

# Application Deployment

Before a cloudlet can act as an offloading site, it needs to be provisioned with the application that serves the mobile client’s requests. This section discusses various general approaches how to deploy an application on another machine. Since a cloudlet does not guarantee to have a connection to the Internet or to other external sources, every 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 minimizes the coupling between source and target system.

## Limitations to Portability

There are certain constraints that need to be met if we want to port an application to another system.

### Instruction Set Architecture

The instruction set architecture on the target machine must be compatible to 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. The same applies if a library that the application depends on is 64-bit.

Since applications often have third-party dependencies that are either 32-bit or 64-bit, we have to request the target system to provide the same instruction set architecture.

### Hardware Dependencies

If the application relies on non-virtualizable special hardware, may it be a minimum number of CPU cores or a camera, the target system has to provide this hardware.

### Software Dependencies

Software often depends on specific versions of other software. The concept of shared libraries allows commonly used software modules; 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 during load or during 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.

We have 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. If a dependency is “missing” depends on the state of the target system, therefore it is difficult to determine which dependencies have to be delivered with the application. E.g., different Linux distributions have different sets of libraries installed; libraries on one system do not necessarily exist on another. There is a large variety of Linux distributions and the user may compile the Linux kernel himself including only the libraries he demands; as well as continuously alter the system. If we relied on the library set of a specific distribution, it would drastically reduce the number of valid target systems. This contradicts the aim of minimizing the mobile device’s assumptions about a cloudlet.

### 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 kind of conflict arises if a software module implements functionality that risks breaking other software. Linux packages such as Debian or Red Hat packages therefore hold metadata, which names dependencies along possible conflicts (cf. [23] [24]).

These two kinds of dependency conflicts are normally avoided by using a package manager, which maintains the system’s state of installed libraries. Therefore, it can fetch dependent libraries and remove conflicting libraries, or forbid the installation of packages which would cause conflicts.

The package manager is a system tool that is eligible to 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 are hindered to 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.

## Source Code versus Binary File Transmission

Transmitting source code instead of executable binary files seems to increase portability at first glance, because it allows compiling straight for the target system. But this benefit can only be achieved if all dependencies’ source code is equally available. 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; but it avoids the time-consuming compilation on the target system.

## Packaging Dependencies

As a result from the arguments in the preluding 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 further than necessary. The following paragraphs discuss alternatives of how to create this file, which we call a package, since it is an archive consisting of numerous files.

### Remote Install

The idea of remote install is to transfer the application along with the software packages that it depends on, and then install these packages on the target system. This is a straightforward approach, but it has some drawbacks. Like mentioned in 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 even may remove its capability to run or install certain other applications. The installation packages must fit the target system, thus making strong assumptions about the cloudlet. Another drawback is the additional time that it takes to install new packages on the target system. Over all, remote install strongly contradicts the goal of not altering the cloudlet state.

### Library Packaging

Instead of including installation packages it is possible to directly include 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. Successful porting from one distribution to a diverse distribution cannot be guaranteed. During this work, we tried to run an application that has been packaged on 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. But it is desirable to have a cloudlet being 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 this application has originally been packaged.

### Static Linking

Static linking allows including the compiled code and all library dependencies in one binary. As soon as the linking succeeded, 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 matters. For example, if A depends on B - i.e. A uses symbols that are delivered by B - the linker needs to link A first. If there are cyclic dependencies, static linking will fail. Another issue is the necessity of static libraries for linking; if only shared libraries are available, 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.

### 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 fashion. Application virtualization uses a similar approach like OS virtualization, i.e. tricking software into acting with a virtual rather than actual existent environment. While OS virtualization emulates the hardware for a guest OS, application virtualization emulates OS functionality for an application. To accomplish this, a runtime intercepts all system calls that an application causes 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 interacts with purely virtual operating system services.

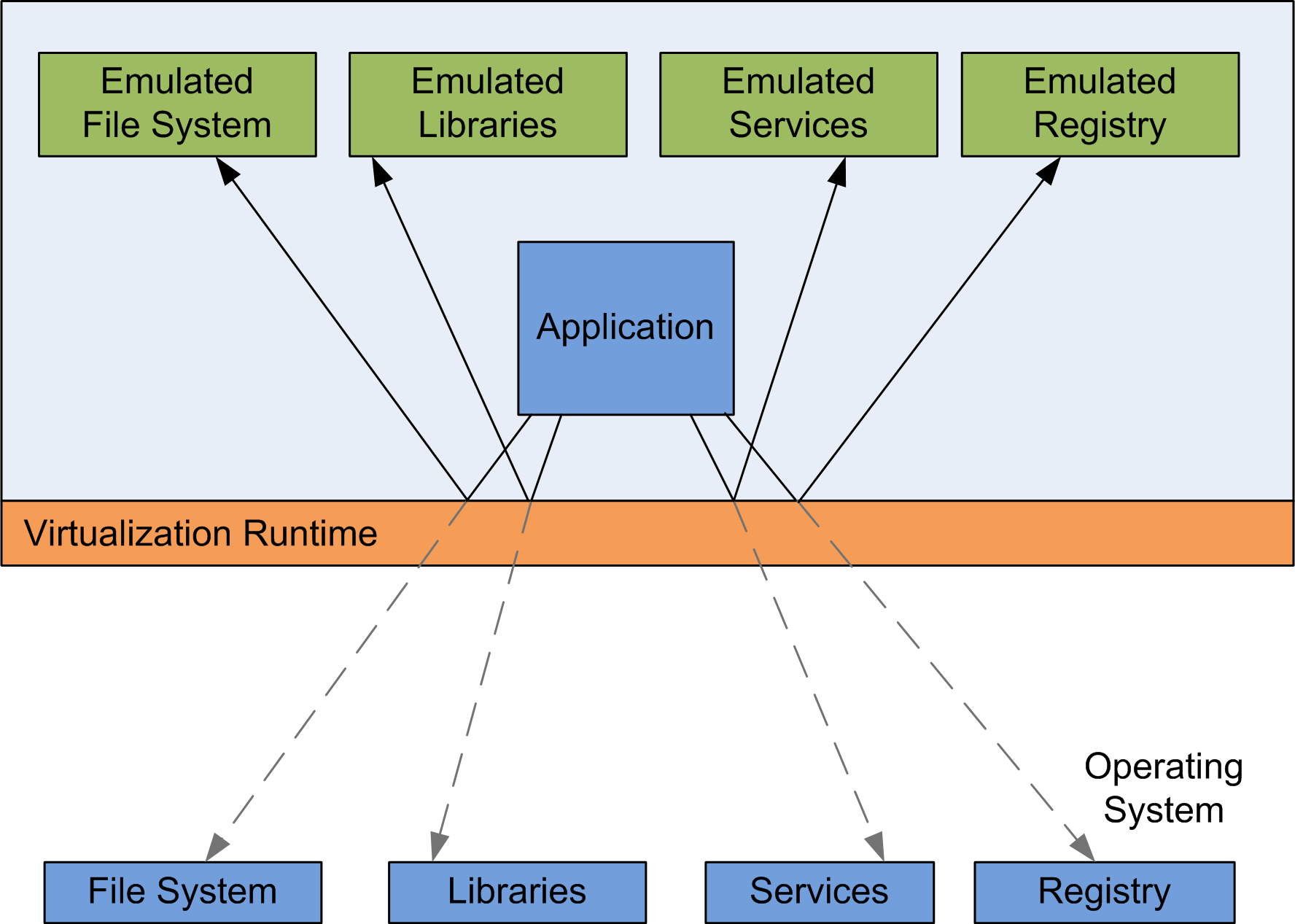


Figure 6: Application Virtualization - System Call Interposition and Redirection

Application virtualization cannot be applied to every kind of application. Device drivers, since they interact with the hardware directly, cannot be virtualized. It is difficult to virtualize software that interacts with the OS internals, such as antivirus programs.

A virtualized application generally executes more slowly than in its non-virtualized form, because every user/kernel mode switch that is caused by a system call results in two further user/kernel mode switches; the first to change from the kernel to the virtualization runtime, the second from the virtualization runtime back to the kernel to realize the modified system call.

Nevertheless, application virtualization offers portability in a degree that the approaches that have been discussed earlier cannot guarantee. Since 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 utilizes application virtualization tools to package the application for its transfer to the cloudlet. The next section gives basic design goals of this implementation along with an introduction to the application virtualization tools that have been used.

# Application Virtualization for Cloudlets

## Design Goals

The cloudlet architecture implementation in this thesis uses application virtualization techniques to support design goals that we chose for our implementation. The main design goals can be described with the words simplicity, generality and quick response.

Simplicity

Setting up a cloudlet should be convenient and accomplished in 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 actual application usage phase has to be seamless. The user must not have to worry about internals like IP addresses and ports.

Generality

Packaged applications should be only loosely coupled to the operating system, therefore should run on many cloudlets. As a consequence, the cloudlet should allow updates and upgrades 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 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 request any code modifications and provides a high degree of application portability. The file size of the virtualized application strongly influences the time until the application is ready for use by the mobile device. Application virtualization packages only dependencies that are necessary for portability, thus keeping the file size small.

## Application Virtualization Tools

We used two tools for creating and executing virtualized applications; one for Linux and one for Windows systems. These two have been chosen because both are freely available and are between the most mature tools among non-commercial solutions.

### CDE

CDE is short for Code, Data and Environment and has been developed as an application virtualizer for Linux by Philip J. Guo and Dawson Engler. The following section introduces CDE and summarizes content from [25].

CDE allows virtualizing applications by monitoring its execution. Through the ptrace system call [26] 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 containing the accessed files is mirrored inside the package. Each time the virtualized application tries to access a file, the corresponding system call gets 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 then 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 from several years in the future” (p.4) [25].

The package can be configured to allow access to specified file paths outside its sandbox. CDE does not guarantee do 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 be accessed, every possible control path has to be examined. This is an undecidable problem; otherwise this would induce that we could predict the program’s outcome without executing it. Especially applications, which implement a plugin structure and load libraries dynamically during runtime, are likely to miss dependencies after virtualization. To face this issue, we can run the CDE packager 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; which is the preferred way when deeper knowledge about the application exists.

### Cameyo

Cameyo [27] is an application virtualizer for Windows. It packages the application and its dependencies into one single .exe file. Different from CDE, which monitors the 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 those snapshots. The second mechanism does not take snapshots but simulates the installation process, keeping track of all of the installer’s actions. This simulated installation does not have any 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 give access to the files outside the package.

# Implementation

In this section, we describe the implementation of our Cloudlet cyber foraging system. First, the basic architecture and the components within it are introduced. Then we take a closer look at the interaction that happens during application offloading. Finally, some chosen implementation details are discussed.

## Basic Architecture

The two basic actors in our architecture are the mobile device and the cloudlet host, which is a machine that lends its resources to the mobile device. All devices are connected to the same wireless network. The cloudlet host runs a hypervisor to host multiple virtual machines. These different VMs provide a selection of various operating system versions. The mobile device may choose one distribution among 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 further relevant properties. This information is collected by the cloudlet client that runs on the mobile device. Every cyber foraging capable application is divided into a client and a server. The client is designed to run on the mobile device while the server is to be offloaded to the cloudlet. When the user decides to start a cyber foraging capable application and the cloudlet client has discovered a suitable cloudlet system, the cloudlet client starts to transmit the server part to the regarding cloudlet server. The server part consists of two items: the application metadata and the application package. The application metadata holds for the offloading process relevant information; the application package is a compressed archive that contains the executable server along with necessary data. After receiving the application package from the mobile cloudlet client, the cloudlet server prepares the retrieved application server for its execution. After signaling the successful start of the application server to the mobile device, the cloudlet client starts the application client, which can then interact with the application server. By offloading resource-intensive tasks to the application server, the mobile device exploits the resource-richness of the cloudlet.

Below, the components that are involved in the previously described process are specified separately.

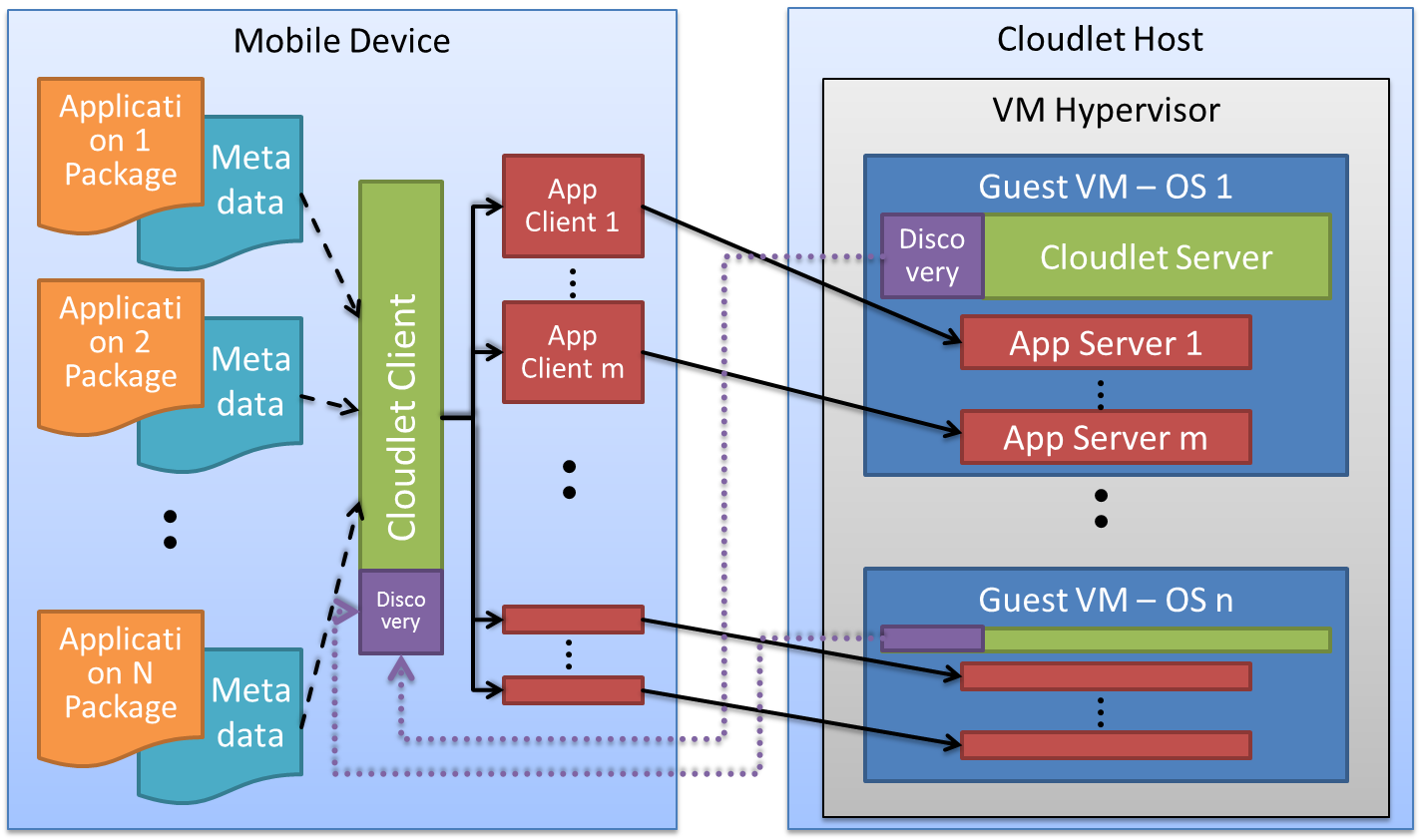


Figure 7: The cloudlet cyber foraging system architecture

### Mobile Device

The mobile device is a multicast supporting device running Android 4.1. All parts of a cyber foraging capable application are stored on the device. This includes the application client and the application metadata as well as the application package, which contains the application server.

### Cloudlet Host

The cloudlet host is a multicast supporting machine that is able to run the virtual machine hypervisor.

### VM Hypervisor

We use KVM, which is a common and mature hypervisor for virtual machines that is part of the Linux kernel. The KVM managed virtual machines connect to the network via the bridged network mode. Thus, each VM obtains its own IP address.

### Cloudlet Client

The cloudlet client is an Android 4.1 application. It searches the mobile device’s storage for cyber foraging capable applications and lists them on the screen. It is also responsible for discovering cloudlet servers. When the user selects to run one of the displayed applications, the cloudlet client transmits the application metadata and application package via HTTP to an appropriate cloudlet server. It visualizes the upload progress and retrieves information from the cloudlet server that is then presented on the screen. After the successful deployment on the cloudlet, the cloudlet client starts the application client, which causes the mobile device to switch to the application’s activity.

### Cloudlet Server

The cloudlet server is a Java program depending on JRE 7 or higher. It embeds a Jetty HTTP server [28] that is responsible for processing file uploads and sending status messages to the cloudlet client. It registers its service as cloudlet server by providing the service information via multicast. Although it is generally portable across operating systems, it relies on external resources for package decompression and terminal execution. When starting the cloudlet server, it detects the underlying operating system automatically and chooses which code to use for the mentioned tasks.

### Discovery

The discovery mechanisms are provided by the JmDNS library [29], which is a pure Java implementation of multicast DNS and the zeroconf framework [30]. The cloudlet server registers a service, thereby publishing information about itself. The cloudlet client uses JmDNS for exploring services that are published on the zeroconf multicast group and adds newly discovered services to its internal list of cloudlet servers.

### Application Client

Each cyber foraging capable 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, providing 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.

### 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 being computed by the application server on its behalf. The computational result is then sent back to the client. For example, the face recognition server mentioned in the cloudlet scenario in 3.3 receives images from the corresponding mobile application client and responds with the faces that it found in that image.

### Application Package

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

### Application Metadata

Every application package is accompanied by a json file providing necessary information about the package. This application metadata describes w

## Application Deployment Sequence

The diagram in Figure 8: Application Deployment on a CloudletFigure 8 visualizes the interaction between the mobile device and the cloudlet taking place during application deployment. The participating actors are the cloudlet client with its JmDNS discoverer and the cloudlet server. After successful deployment, the application client (“:Activity”) starts its interaction with the application server (“:Process”). All mobile to cloudlet interaction uses HTTP requests and responses. The protocol used between the application client and application server depends solely on the implementation of the application.

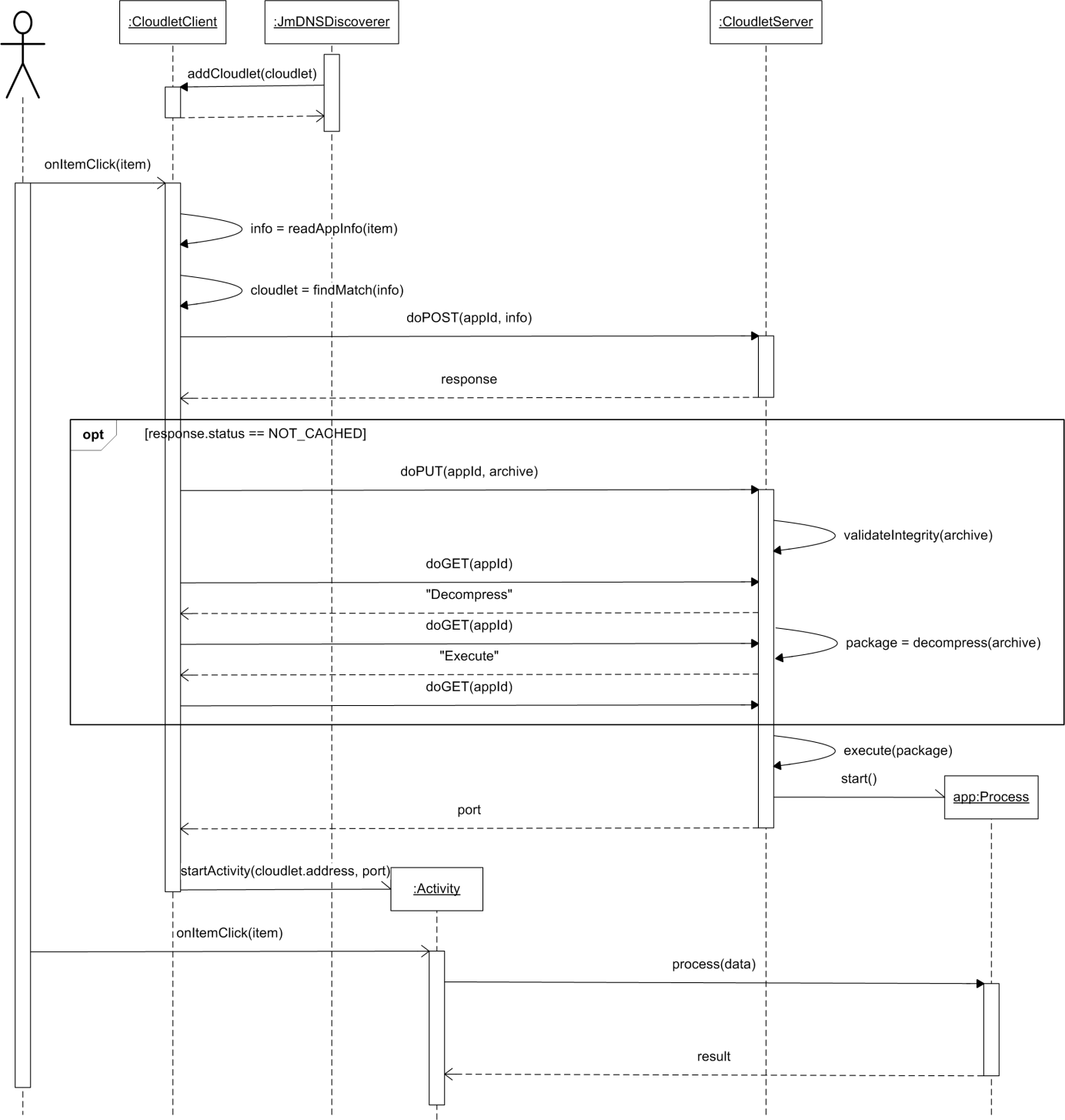


Figure 8: Application Deployment on a Cloudlet

1. The JmDNSDiscoverer finds a new Cloudlet service and adds the accompanied information to the CloudletClient’s queue of available cloudlet servers.
2. The user clicks on an item in the list of applications on his mobile device. The application behind this item is to be offloaded to a cloudlet.
3. The application metadata is retrieved and a cloudlet that matches the application requirements is found in the queue of available cloudlet servers.
4. The cloudlet client issues an HTTP Post request to the CloudletServer. This request holds the unique application identifier and the application metadata.
5. If an application with this identifier is already deployed on the cloudlet, continue with step 10.
6. The application is not cached on the cloudlet, therefore the CloudletClient transmits the application package (“archive”) to the CloudletServer. On the cloudlet, the application package’s integrity is validated by comparing the md5 checksum and file size to the values listed in the application metadata.
7. The CloudletClient starts to listen for CloudletServer messages by sending a GET message to the CloudletServer. It immediately sends a new request after receiving a response, thus allowing the CloudletServer to push messages to the CloudletClient.
8. The CloudletServer decompresses the application package archive and informs the CloudletClient about his activity.
9. The “Execute” progress status is sent to the CloudletClient.
10. The CloudletServer starts a new system process for the application server.
11. The CloudletServer sends the port on which the application server operates to the CloudletClient.
12. The CloudletClient now starts the application client activity with the given port and cloudlet address.
13. The application client (“:Activity”) sends data to the application server (“:Process”) . The application server processes this data and returns the result to the application client on the mobile device, i.e. cyber foraging is taking place.

## Details

This section aims to give the reader a deeper insight into the implementation done in the context of this thesis. To achieve this intention, selected design decisions are shown and discussed.

### Cloudlet Server Package Architecture

The cloudlet server code turns a virtual machine into a functioning task offloading site. It is completely written in Java 7 to ensure execution on a plurality of operating systems. The code is divided into different packages with each one fulfilling a single part of the cloudlet server’s tasks. The server package holds classes that accept requests from the cloudlet client and respond with the demanded action. This functionality is provided by embedding a basic Jetty HTTP server that maps HTTP requests to the corresponding HTTP Servlets which process the client’s demands. For service discovery, the jmdns package allows to register the cloudlet service in the network through usage of JmDNS, which is a Java implementation of the zeroconf networking techniques. The knowledge how to process uploaded applications is bundled within the packagehandler package. This package contains interfaces and abstract classes that have to be implemented operating system and file type specific. For this thesis, Windows and Linux packagehandlers have been realized. The server and packagehandler packages both use the fileprocessing package, which is able to copy and delete files, compute checksums and decompress archive files. Breaking the cloudlet server into the previously described packages aims to facilitate further maintenance and extensibility.

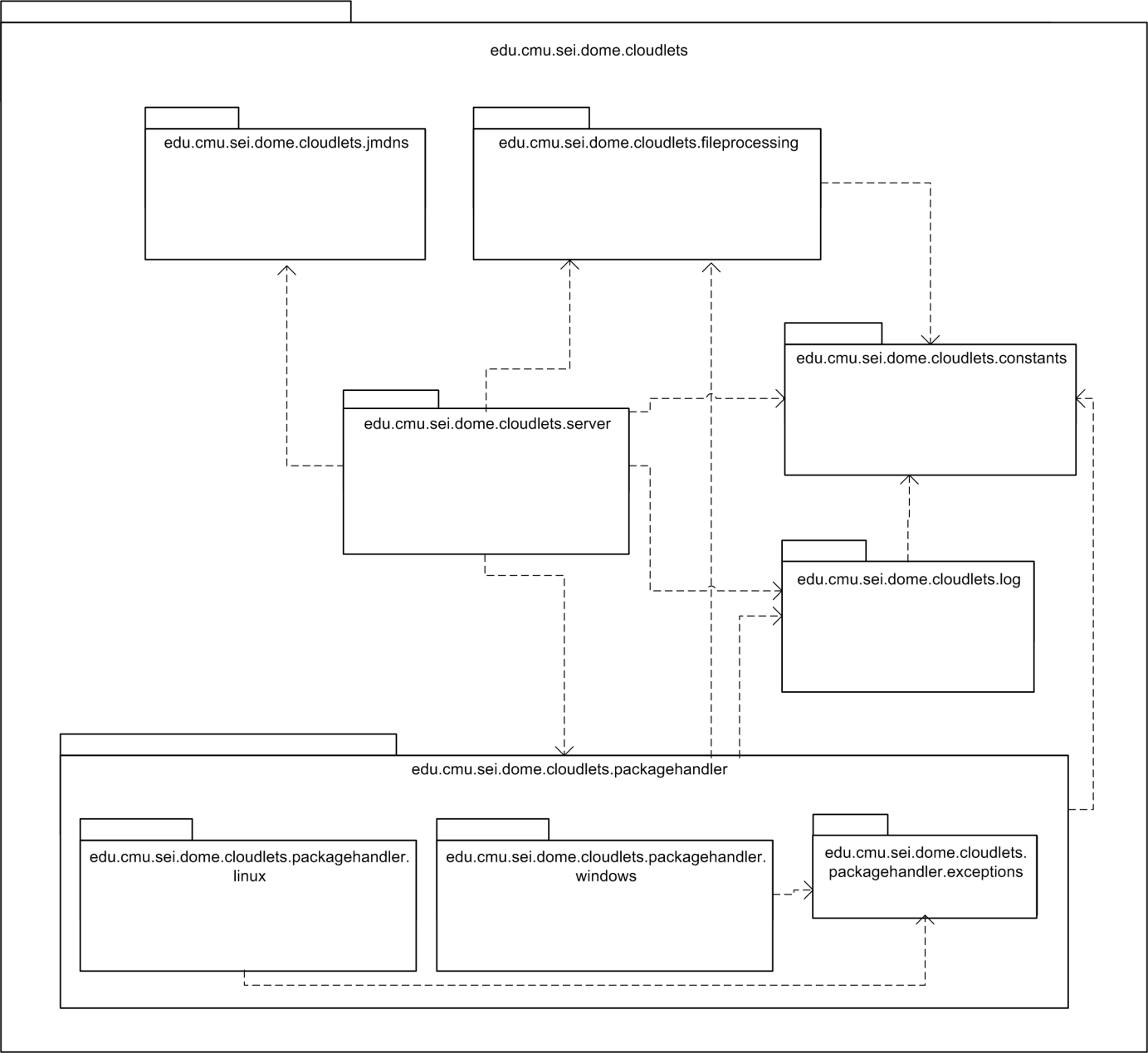


Figure 9: Cloudlet Server Package Architecture

### Application Metadata and Cloudlet Requirements Matching

The application metadata, which accompanies each application package, provides information that is necessary for the cyber foraging process. It is a JSON file whose structure is described by the schema in **Fehler! Verweisquelle konnte nicht gefunden werden.**. The name and description fields give 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 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 legitimate cloudlet properties. The properties may be arbitrary key-value pairs or minimum or maximum numeral requirements, e.g. cores\_min: 4",

"items": {

"type": "object"

}

}

}

}

Listing 1: 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 a possible set of requirements that need to be satisfied by a cloudlet. 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. The properties are published via the JmDNS service registration, enabling the cloudlet client to find a matching cloudlet server. Cloudlet properties match a set of requirements if and only if

1. every field in the set can be found in the cloudlet properties with the same value, except for:
2. a number field <name>\_min must be met by a cloudlet field <name> with a number greater or equal
3. 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 **Fehler! Verweisquelle konnte nicht gefunden werden.**. The given cloudlet properties match the second set of requirements, therefore it is a legitimate offloading site for the application server.

|  |  |
| --- | --- |
| "cloudlet": [  {  "os": "windows xp",  "architecture": "x86"  },  {  "os": "windows 7",  "architecture": "x86-64",  "cores\_min": 4  }  ] | {  "os": "windows 7",  "architecture": "x86-64",  "cores": 8  } |
| Sets of cloudlet requirement | Cloudlet properties |

Listing 2: Cloudlet Requirements Matching

### RESTful Architecture

Representational State Transfer, also known as REST, 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 utilizes the HTTP methods GET, PUT, POST and DELETE. While PUT and DELETE are idempotent, i.e. a second call does have no further effect, GET may not have any side effect. How these requests on a resource are handled in detail is hidden from the client. There may be any intermediates between the client and the actual server, which are not visible for the client that addresses resources rather than servers.

Our cloudlet implementation takes advantage of the REST principles to provide an easy understandable pattern for application management on the cloudlet. The resources in our case are the application servers; they can be addressed via the checksum of the application package. Given the cloudlet server’s address and port, the address for one resource is http://<address>:<port>/apps/<checksum>. This address scheme should provide each resource with a unique identifier; this goal is only almost accomplished. One application’s md5 hash may theoretically collide with another, since md5 is no longer considered to be safe. A more advanced cryptographic hash function could minimize this risk. The following table shows the effect of each HTTP request on a resource in the context of our cloudlet server implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **GET** | **POST** | **PUT** | **DELETE** |
| Get a message naming the current status of the application. | Transmit the application metadata JSON file to create an entry for the application on the cloudlet. | Replace the current representation of the application with the transmitted application package or create it, respectively. | Delete the representation and entry of the application. |

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

### 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, we would like the server to inform the client about the application deployment progress. Since the server side determines the actual deployment status, the server should be able to determine the time the progress information arrives at the client. Each status change notification should be as prompt as possible.

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 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 again held by the server until it decides to respond.

In our cloudlet implementation 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 gets 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.

@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 3: Client Long Polling - EventListener.java

The server counterpart is the RESTservlet and PushHandler classes, which encapsulate the request and suspend or continue it, respectively.

@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 4: Server Long Polling: RESTservlet.java

**private** **void** pushToClient(String message, **int** status) **throws**

IOException {

**if** (message == **null** || message.equals(""))

**return**;

Continuation continuation = waitForClientRequest();

**if** (continuation == **null**)

**return**;

Log.*println*(appId, "Respond: " + message);

HttpServletResponse resp = (HttpServletResponse) continuation

.getServletResponse();

resp.setContentType("text/html");

resp.setStatus(status);

resp.getWriter().write(message);

continuation.complete();

}

Listing 5: Server Long Polling - PushHandler.java

### 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 [31] 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 gives the possibility to create an abstraction hierarchy 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 10 visualizes the entire PackageHandler Bridge as a UML class diagram.

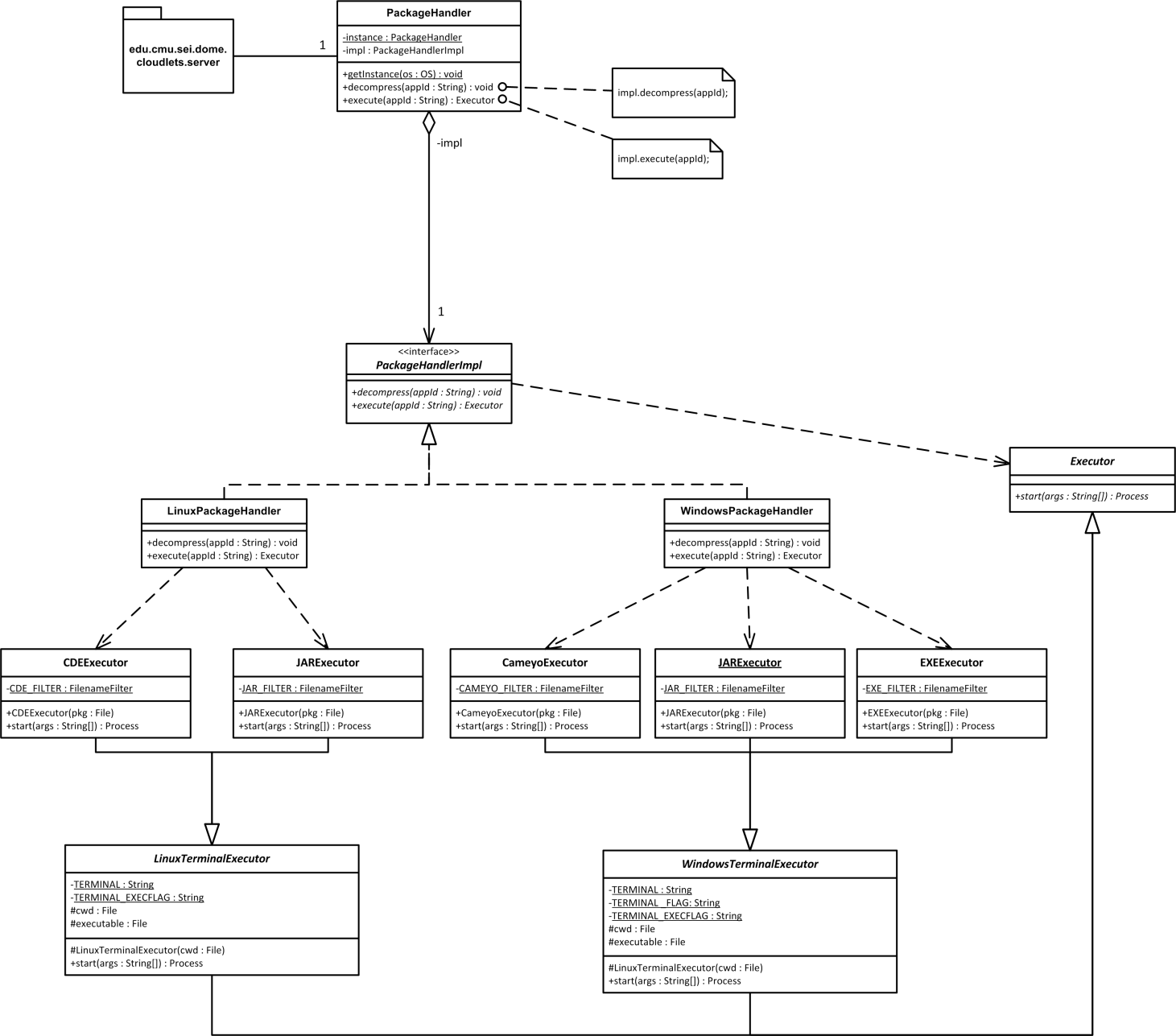


Figure 10: Bridge Pattern decoupling PackageHandler Abstraction from OS specific Implementation

# Evaluation and Comparison with VM Synthesis

## 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 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 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.

## Quantitative Analysis

To identify the battery efficiency and speed of our 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 found 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. We use the NULL application to determine the baseline for transmission overhead and battery consumption.

Table 2: File Sizes of Applications and Compressed Application Packages

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Object Recognition** | | **Speech Recognition** | | **Face Recognition** | | **NULL** | |
| Application size (MB) | 25.340 | | 100.140 | | 34,449 | | 0.009 | |
| Compressed Package size (MB)  (cde | cameyo) | 28.492 | - | 67.748 | 65,370 | - | 13.090 | 1.133 | 0.940 |

### 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. 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 [32].

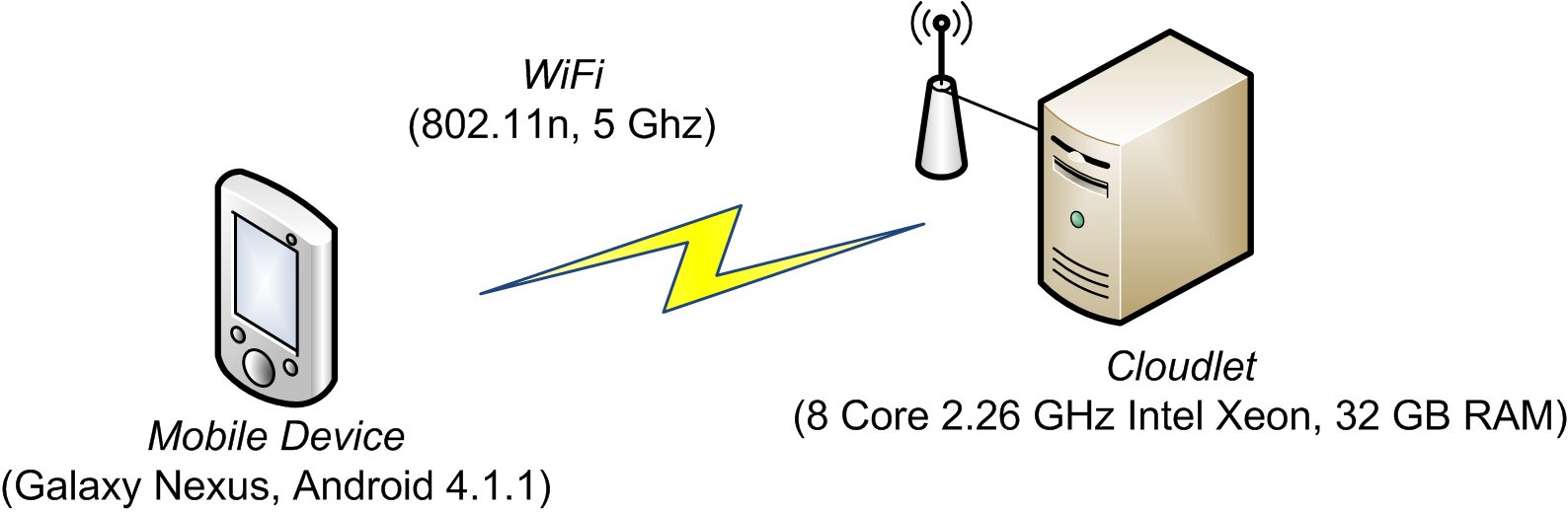


Figure 11: Evaluation Experiment Configuration

Table 3 and Figure 12 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 12: 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 most 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.

### 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 cloudlet form the first group; Cameyo applications for Windows XP form the latter.

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

and

.

Therefore, we suppose a linear dependence between package size and deployment time. An explanation for this observation is the obvious linear dependence between file size and file transmission time, which is mostly determined by the Wi-Fi bandwidth, and 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 12).

Linear regression indicates the following relations:

CDE/Ubuntu 10.04:

Cameyo/WinXP:

Figure 13: Application Package Size in relation to Deployment Time

In 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 we know that application transmission time is in linear relation to package size, there is a linear dependence between package size and energy consumption.

The sample correlation coefficients are:

Linear regression:

CDE/Ubuntu 10.04:

Cameyo/WinXP:

Figure 14: Application Package Size in relation to Energy Consumption

### Comparison with VM Synthesis

The VM synthesis cloudlet reference architecture has been evaluated in [8]. Like in our 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. If we take the numbers from the revised VM synthesis prototype evaluation (p. 18ff) [8] and apply a linear regression analysis, we find a relation of and . Whereby ““ is the total amount of data, i.e. the sum of disk image overlay size and memory overlay size.

As a consequence of our 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 we created disk and memory overlays of the same applications that we used for evaluating the application virtualization implementation. Hereby, 64-bit versions of Ubuntu 12.04 and Windows XP with Service Pack 2 were used. We find 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 |

## Qualitative Analysis

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

### 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 is comprehensible since 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 offloading. 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 is useless. A possible workaround is to let the mobile device transmit the complete final VM image, though 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, it will lead to a looser coupling compared with application virtualization because the cloudlet only needs to run a hypervisor that can handle the transferred images.

### Patchability of the Target System

In the context of VM synthesis, base VMs cannot be updated without invalidating every overlay that refers 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 supposedly correlates with the number of applications that support VM synthesis cloudlets. As the former number grows, the cloudlet infrastructure will get more scattered. There might be outdated base VMs which are still kept for legacy reasons. Such will build an infrastructure where the mobile client relies on the cloudlet to provide a particular resource. 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 is 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.

### Range of Offload Ready Applications

Like the authors of CDE explain, applications that depend on specialized hardware or device drivers (p.7) [25] 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 [33]. 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 does not run within a sandbox. This means that it is not portable across system boundaries; the main goal is rather to avoid an 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. Applications that use VM synthesis as distribution mechanism also expect the cloudlet to have specialized hardware if demanded, though.

### 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 essential to a mission’s success.

VM synthesis simply mirrors the application’s original functionality because it reconstructs the entire operating system under which the application has been installed. If this installation has been faultless, the offloaded application should be correct as well. Like mentioned in the previous section, special hardware requirements may have to be provided by the cloudlet. So these requirements have to be documented by the application 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.

### Application Preparation Overhead

Preparing an application for deployment on a cloudlet should require only low effort. VM synthesis and application virtualization do both not require any modifications or 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. This is a convenient mechanism and the main difficulty lies in the regular installation.

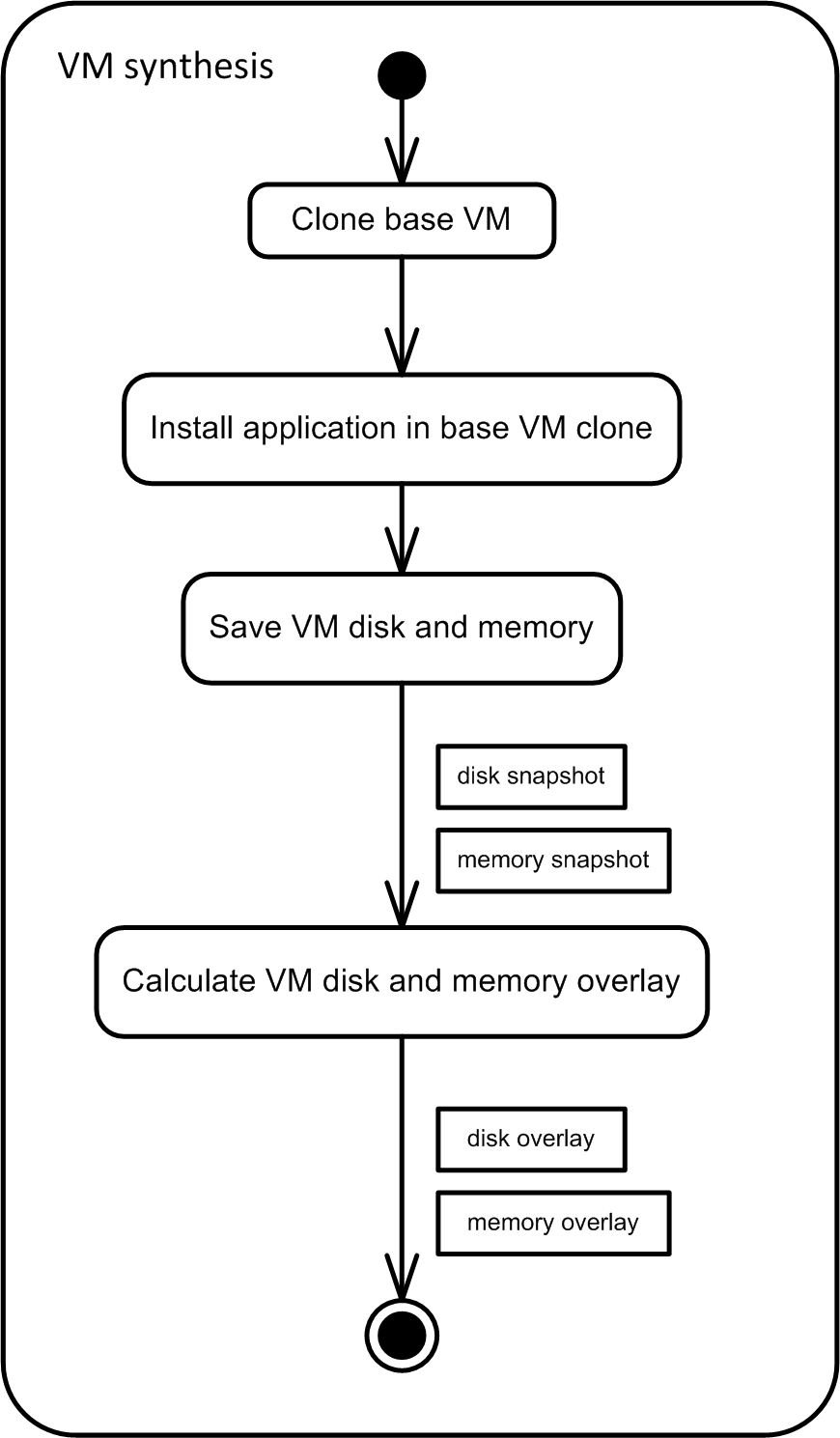


Figure 15: VM overlays Creation Process

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 modifying the 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.

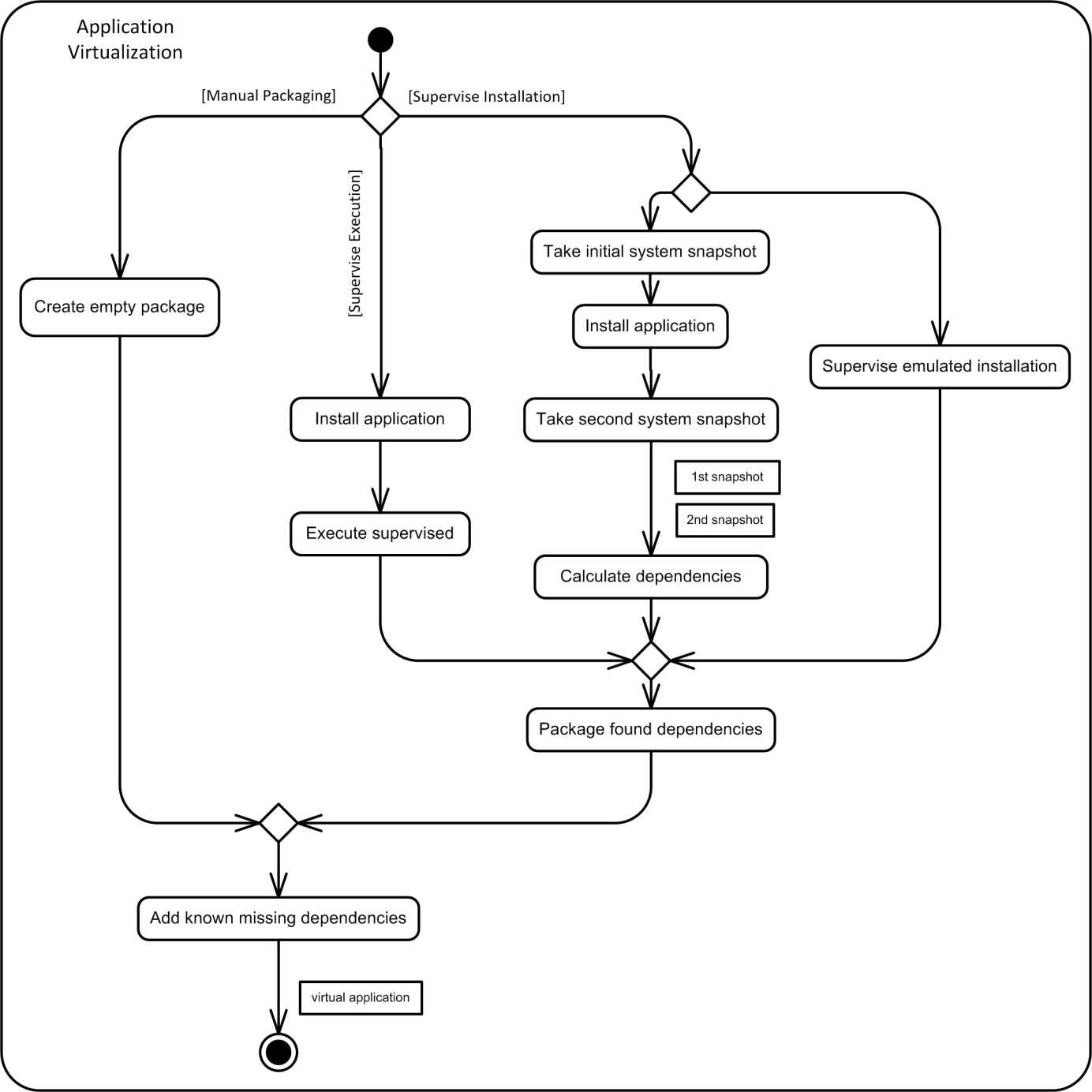


Figure 16: Virtual Application Creation Process Alternatives

### Operation Overhead

For running an application server in the cloudlet implementation that we proposed in section 6, we embed the server into an application virtualization runtime environment which in itself runs on a virtual machine. 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 least slowdowns whereby I/O-intensive tasks had the largest slowdowns (p. 13) [25].

Not only influences the application virtualization runtime environment the execution performance but also the hardware virtualization layer that enables us to execute the virtualized application on a virtual machine rather than on the native OS. Hosting a guest OS within a virtual machine 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 [34] 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.

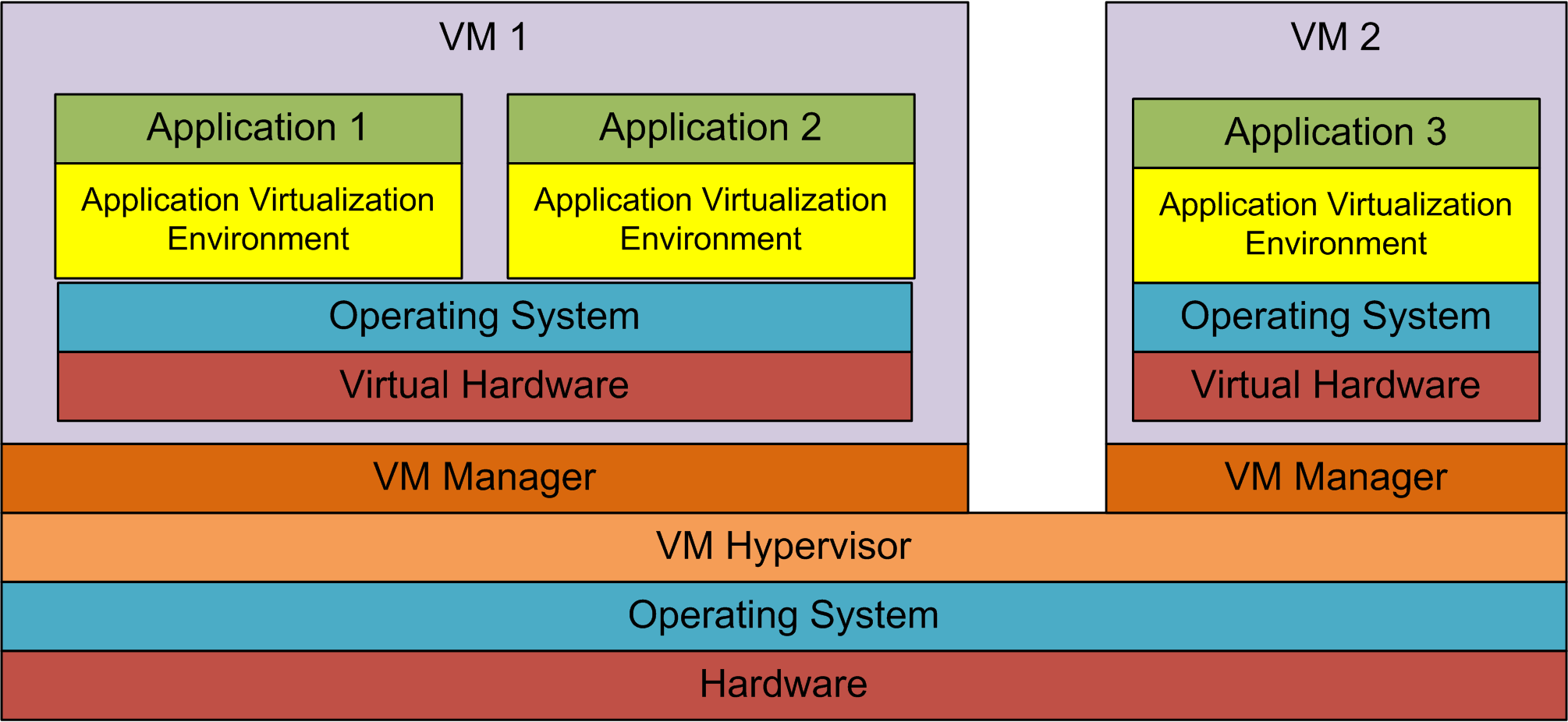


Figure 17: Application Virtualization Layer Architecture

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 virtual machines than our application virtualization based solution. Consider Linux applications and Windows applications: the VM synthesis implementation would have to manage 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 17 and Figure 18). 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. Since a VM’s workload is much larger than the actual application’s workload, a context switch is very costly. We consider, for example, VMs where each uses 4 GB of memory and a cloudlet host with 32 GB of memory, which corresponds to our test scenario in section 7.2. Hence, running many applications soon exceeds the host’s memory because every application runs on separate virtual machine. 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 like e.g. Ubuntu for only one single application includes system functionality that is not necessary for the specific application and adds to the computational overhead.

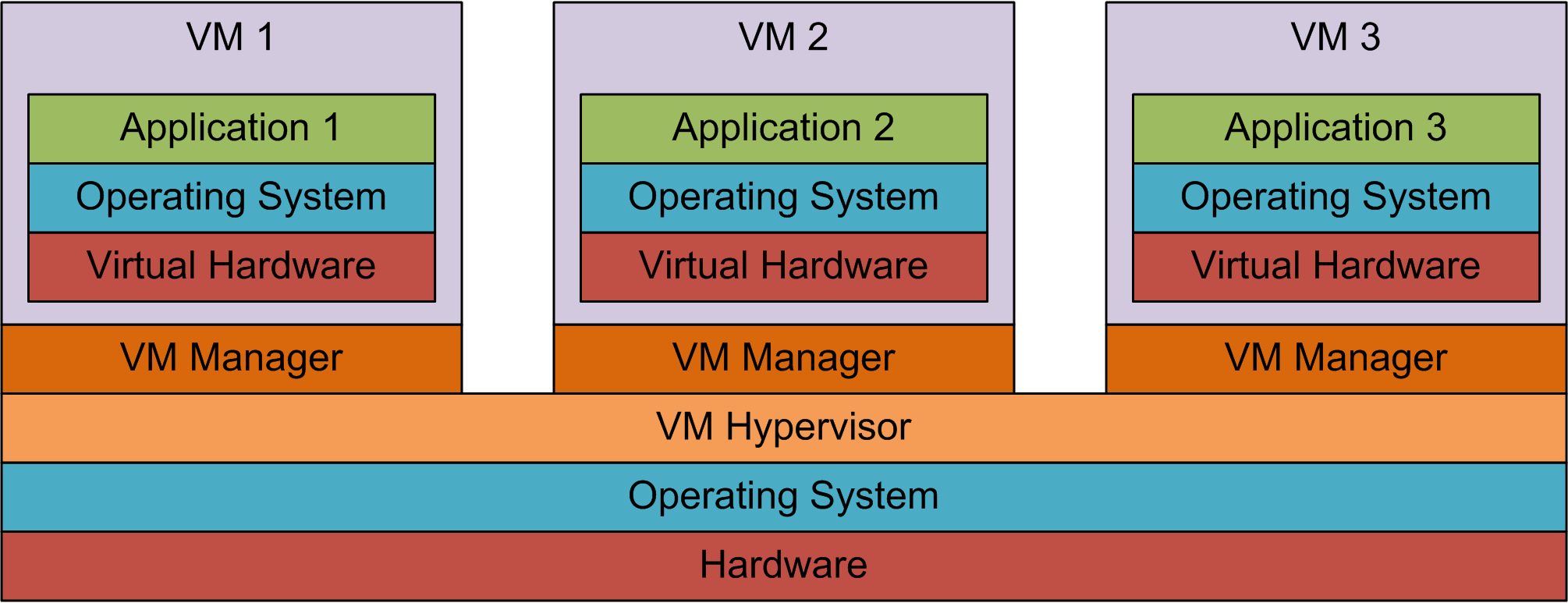


Figure 18: VM Synthesis Layer Architecture

### 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 within 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 virtual machine 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 virtual machines 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 virtual machine, 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) [35]. Therefore, it is the hypervisors task to shut down misbehaving VMs.

Our application virtualization cloudlet solution runs applications that need the same operating system family on the same VM. If we aim to provide load balancing in the future, we may distribute them on separate VMs. But still applications run simultaneously on the same guest OS; this requires further isolation mechanisms because they share the same memory, disk and other system utilities. Because we run virtualized applications, they are embedded into a runtime environment that isolates them not only from the guest OS but also from each other. CDE and Cameyo both use sandboxing techniques which virtualize system resources such as the file system. Sandboxing uses a lower degree of isolation and is therefore not as secure as separation through virtual machines. CDE runs the packaged application within a chroot jail [36], thus preventing it to access files outside its package. This sandbox is, however, vulnerable to attacks that break the isolation mechanism [37]. Consequently, VM synthesis offers better isolation between applications.

# Related Work

The idea of encountering mobile devices’ resource-poverty through leveraging external resources, termed as cyber foraging [7], has been followed by intensive research on this topic. Various cyber foraging systems have been developed, which differ in terms of the strategy they apply to benefit from remote resources.

One such 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 determines when to offload code to the remote machine. Examples of such cyber foraging systems are Spectra [12], Chroma [13] [14] [15], MAUI [17] and Scavenger [16].

CloneCloud [18] follows the same principle but automatically partitions code at thread level without need for manual marking.

Another cyber foraging strategy is to offload an entire application. Goyal and Carter [21] let the mobile device trigger the remote download and installation of applications on an external virtual machine. This approach is closely related to the work presented in this thesis. However, the work in this thesis uses so-called cloudlets as offloading site, which do not rely on Internet access. Furthermore, the cloudlet shall not be altered by extending it through remote installations because this may lead to dependency conflicts or overweight systems. Instead, application virtualization eliminates the need for durable installation.

The cloudlet architecture, which is also the setting for the cyber foraging implementation presented in this thesis, has been described in [1] and [5]. Offloading takes place by establishing a virtual machine 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 named VM synthesis is implemented [1] [5] [8]. The mobile device carries a so-called overlay which 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 [5].

The work in this thesis also uses the cloudlet architecture mentioned above and focuses on providing external resources to mobile devices in hostile environments. Instead of applying the VM synthesis strategy, it explores the applicability of application virtualization for cyber foraging. During this thesis a corresponding architecture has been implemented whose characteristics are evaluated against VM synthesis.

# Limitations and Future Work

Application virtualization loses its advance in portability when the application that shall be virtualized is restricted to particular hardware or drivers. Requiring a very particular environment opposes the idea of general purpose cloudlets.

A risk in using application virtualization is to miss out dependencies. Because it is impossible to automatically detect all dependencies, human knowledge of the application is required. Especially applications that provide a plug-in architecture are likely to miss dependencies when being virtualized without manual intervention. The application virtualization tools used in this thesis both allow manual dependency adding to complete a virtual package. Future work should focus on how to facilitate 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; an assisting tool may help by suggesting typically used components for inclusion.

The implementation architecture in this thesis does not allow running application servers on the same cloudlet if they use the same fixed port number. Some kind of virtualization has to be introduced that decouple fixed port numbers from the actual ports provided by the cloudlet.

Because sandboxed applications still share common resources such as port numbers, they may conflict with each other. Future work may analyze the overhead of isolating each application into its own virtual machine like is done in VM synthesis.

A real-world cloudlet solution has to satisfy security demands, which have been excluded in this thesis. Therefore, the implementation should be extended with trust establishment mechanisms between mobile device and cloudlet.

The presented mechanism for discovering a cloudlet that fits the application’s requirements is rather primitive because it assumes the cloudlet and mobile device to use the same names 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 while cyber foraging takes place. This requires live migration, i.e. resuming the halted application on another cloudlet while preserving computational state and minimizing downtimes. Migration for 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 availability of cyber foraging. A cloudlet may support both with application virtualization being the preferable because faster strategy. As a fallback strategy in case of unsatisfied system requirements an entire virtual machine image, or if a correspondent base VM is available, an overlay may be transferred. The combination of VM synthesis and application virtualization needs further exploration.

# Conclusion

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

In this thesis, we have focused on cyber foraging in so-called 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 offloading site is taken 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 [1] [5] [8] for cloudlet provisioning has been the incitement for this thesis.

The first part of this thesis explains the concept of cyber foraging in general and cloudlet-based cyber foraging in hostile environments in particular. Different strategies to cyber foraging are discussed and requirements for cloudlet-based cyber foraging are presented. We have explored application virtualization as a strategy for cyber foraging. So the second part of this thesis gives 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 utilizes application virtualization has been implemented. This implementation is introduced by giving an architectural overview and discussing selected implementation decisions. Subsequently, the implementation has been evaluated and the achieved results and characteristics have been compared to the VM synthesis strategy. The final part identifies limitations that have not yet been overcome.

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 time and energy consumption. We have shown that these are linear dependent from 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 each member of an appropriate operating system family suffices 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 patches without invalidation of the relationship between cloudlet and application.

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

Nevertheless, application virtualization is a promising strategy for cyber foraging in resource-constrained environments because of its relative lightweightedness and high portability. It is not only an alternative to VM synthesis but can also be a valuable addition in a combined cyber foraging model. Further work may build on the implementation in this thesis and try to overcome some of its shortcomings.

# References

|  |  |
| --- | --- |
| [1] | 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. |
| [2] | "Siri," Apple, [Online]. Available: http://www.apple.com/iphone/features/siri.html. [Accessed 31 07 2012]. |
| [3] | "Google Goggles," Google, [Online]. Available: http://www.google.com/mobile/goggles/. [Accessed 31 07 2012]. |
| [4] | Wikipedia, "Frame Rate," [Online]. Available: http://en.wikipedia.org/wiki/Frame\_rate. [Accessed 7 8 2012]. |
| [5] | K. Ha, G. Lewis, S. Simanta and M. Satyanarayanan, *Cloud Offload in Hostile Environments,* Pittsburgh, PA, USA: Carnegie Mellon University, 2011. |
| [6] | 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]. |
| [7] | M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications,* pp. 10-17, August 2001. |
| [8] | 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. |
| [9] | 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]. |
| [10] | Object Management Group (OMG), *CORBA Interface Definition Language Specification, Version 3.5,* 2011. |
| [11] | Google, "Google Maps API Web Services," Google Inc., [Online]. Available: https://developers.google.com/maps/documentation/webservices/. [Accessed 6 8 2012]. |
| [12] | 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. |
| [13] | 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. |
| [14] | 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. |
| [15] | 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. |
| [16] | M. D. Kristensen, *Empowering Mobile Devices Through Cyber Foraging,* Aarhus, Denmark: Aarhus University, Department of Computer Science, 2010. |
| [17] | 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. |
| [18] | 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. |
| [19] | 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. |
| [20] | 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]. |
| [21] | 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. |
| [22] | 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]. |
| [23] | 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]. |
| [24] | "RPM Packager Documentation: Dependencies," 22 09 2005. [Online]. Available: http://www.rpm.org/wiki/PackagerDocs/Dependencies. [Accessed 18 09 2012]. |
| [25] | P. J. Guo and D. Engler, "CDE: Using System Call Interposition to Automatically Create," in *Proceedings of the 2011 USENIX Annual Technical Conference*, Portland, OR, USA, 2011. |
| [26] | "ptrace (2) - Linux man page," [Online]. Available: http://linux.die.net/man/2/ptrace. [Accessed 14 8 2012]. |
| [27] | Cameyo, "Cameyo User Guide," 6 2012. [Online]. Available: http://cameyo.com/doc/CameyoManual.pdf. [Accessed 15 8 2012]. |
| [28] | "Jetty," 2012. [Online]. Available: http://www.eclipse.org/jetty/. [Accessed 18 09 2012]. |
| [29] | A. van Hoff, R. Blair and P. Frisch, "JmDNS," 25 08 2011. [Online]. Available: http://jmdns.sourceforge.net/. [Accessed 18 09 2012]. |
| [30] | "Zero Configuration Networking (Zeroconf)," [Online]. Available: http://www.zeroconf.org/. [Accessed 18 09 2012]. |
| [31] | E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Bridge," in *Design Patterns*, Addison-Wesley, 1995, pp. 151-161. |
| [32] | Monsoon Solutions Inc., "Power Monitor," 2008. [Online]. Available: http://www.msoon.com/LabEquipment/PowerMonitor/ . [Accessed 13 09 2012]. |
| [33] | "Latest Cameyo 2," Cameyo Blog, 25 04 2012. [Online]. Available: http://cameyoco.blogspot.com/2012/04/latest-cameyo-2.html. [Accessed 06 09 2012]. |
| [34] | 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]. |
| [35] | 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. |
| [36] | 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]. |
| [37] | 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]. |