Date of acceptance Grade

Instructor: D.Sc. Miika Komu
Supervisor: Prof. Sasu Tarkoma

HIP based mobility for Cloudlets

Juhani Toivonen

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, Wednesday 11th November, 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Juhani Toivonen

Työn nimi — Arbetets titel — Title
HIP based mobility for Cloudlets

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	Wednesday 11th November, 2015	69 pages + 8 appendix pages

Tiivistelmä — Referat — Abstract

Computation offloading can be used to leverage the resources of nearby computers to ease the computational burden of mobile devices. Cloudlets are an approach, where the client's tasks are executed inside a virtual machine (VM) on a nearby computing element, while the client orchestrates the deployment of the VM and the remote execution in it.

Mobile devices tend to move, and while moving between networks, their address is prone to change. Should a user bring their device close to a better performing Cloudlet host, migration of the original Cloudlet VM might also be desired, but their address is then prone to change as well. Communication with Cloudlets relies on the TCP/IP networking stack, which resolves address changes by terminating connections, and this seriously impairs the usefulness of Cloudlets in presence of mobility events.

We surveyed a number of mobility management protocols, and decided to focus on Host Identity Protocol (HIP). We ported an implementation, *HIP for Linux* (HIPL), to the Android operating system, and assessed its performance by benchmarking throughput and delay for connection recovery during network migration scenarios.

We found that as long as the HIPL hipfw-module, and especially the Local Scope Identifier (LSI) support was not used, the implementation performed adequately in terms of throughput. On the average, the connection recovery delays were tolerable, with an average recovery time of about 8 seconds when roaming between networks. We also found that with highly optimized VM synthesis methods, the recovery time of 8 seconds alone does not make live migration favourable over synthesizing a new VM.

We found HIP to be an adequate protocol to support both client mobility and server migration with Cloudlets. Our survey suggests that HIP avoids some of the limitations found in competing protocols. We also found that the HIPL implementation could benefit from architectural changes, for improving the performance of the LSI support.

ACM Computing Classification System (CCS):
C.2.1 (Network Architecture and Design), C.2.2 (Network Protocols), C.2.4 (Distributed Applications)

Avainsanat — Nyckelord — Keywords
Cloudlet, Host Identity Protocol, HIP for Linux, Cyber foraging, Migration

Säilytyspaikka — Förvaringsställe — Where deposited
University of Helsinki E-Thesis: `https://ethesis.helsinki.fi/`

Muita tietoja — Övriga uppgifter — Additional information

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Juhani Toivonen

Työn nimi — Arbetets titel — Title
HIP based mobility for Cloudlets

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	11. marraskuuta 2015	69 sivua + 8 liitesivua

Tiivistelmä — Referat — Abstract

Liikkuvassa tietojenkäsittelyssä laskennan ulkoistaminen on menetelmä, jolla voidaan käyttää ympäristössä olevien tietokoneiden resursseja keventämään mobiililaitteeseen kohdistuvaa laskennallista rasitusta. Cloudletit ovat eräs ratkaisu mobiililaskennan ulkoistamiseen, jossa laitteessa suoritettavia tehtäviä siirretään suoritettavaksi tietokoneessa ajettavaan virtuaalikoneeseen. Mobiililaite ohjaa virtuaalikoneen luomista ja siinä tapahtuvaa laskentaa verkon yli.

Mobiililaitteen taipumus liikkua käyttäjänsä mukana aiheuttaa haasteita nykyisen TCP/IP protokollapinon joustavuudelle. Mobiililaitteen siirtyessä verkosta toiseen, on tyypillistä että sen IP-osoite vaihtuu. Mikäli mobiililaite siirtyy lähelle Cloudlet-isäntäkonetta, joka olisi resurssiensa ja tietoliikenneyhteyksiensä puolesta suotuisampi käyttäjän tarpeisiin, voi käyttäjän Cloudlet-virtuaalikoneen siirtäminen olla toivottavaa. Tällöin kuitenkin myös virtuaalikoneen osoite voi vaihtua. TCP/IP ratkaisee osoitteen vaihtumisen katkaisemalla yhteyden, mikä käyttäjien liikkuvuutta rajoittavana tekijänä tekee Cloudlet-ratkaisun käytöstä vähemmän houkuttelevaa.

Tässä tutkielmassa tutustuimme joukkoon sopivaksi arvioimiamme liikkuvuutta tukevia protokollia, ja valitsimme niistä HIP -protokollan lähempää tarkastelua varten. Teimme *HIP for Linux* -protokollaohjelmistosta sovituksen Android-käyttöjärjestelmälle ja tutkimme sen soveltuvuutta liikkuvuuden tukemiseen mittaamalla sen avulla muodostetuilla yhteyksillä saavutettavia siirtonopeuksia sekä yhteyden palautumiseen kuluvaa aikaa osoitteenvaihdosten yhteydessä.

Mikäli HIPL:in hipfw-moduuli, ja erityisesti sen LSI-tuki (IPv4-sovellusrajapinta) ei ollut käytössä, mittaustemme mukaan protokollatoteutus suoriutui Cloudlet-käyttöön riittävän hyvin siirtonopeuksien suhteen. Lisäksi yhteyksien palauttaminen osoitteenvaihdosten yhteydessä sujui siedettävässä ajassa, keskimäärin noin kahdeksassa sekunnissa. Hyvin optimoitujen Cloudlet-virtuaalikoneiden synteesimenetelmien vuoksi kahdeksan sekunnin toipumisaika yksinään ei tarjoa virtuaalikoneen siirtämisestä merkittävää etua uuden luomiseen nähden.

HIP protokolla soveltuu yhteydenpitoon sekä mobiililaitteesta Cloudlet-isäntäkoneille, että Cloudlet-virtuaalikoneeseen; pienehkön kirjallisuuskatsauksen perusteella muita oleellisia protokollia hieman paremmin. Tunnistimme myös uudistamistarpeen HIPL-toteutuksen arkkitehtuurissa LSI-tuen suorituskyvyn parantamiseksi.

ACM Computing Classification System (CCS):
C.2.1 (Network Architecture and Design), C.2.2 (Network Protocols), C.2.4 (Distributed Applications)

Avainsanat — Nyckelord — Keywords
Cloudlet, Host Identity Protocol, HIP for Linux, Cyber foraging, Migration

Säilytyspaikka — Förvaringsställe — Where deposited
Helsingin yliopiston E-Thesis: `https://ethesis.helsinki.fi/`

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Cloudlets [60] bring additional computational power close to the users, and allow them to execute arbitrary applications inside a Virtual Machine (VM). The closer the Cloudlets are to the users, the better the performance and the user experience [9]. However, Cloudlets base their communications on the TCP/IP networking stack, and hence, their support for mobility, i.e., roaming between different networks, is limited.

In Internet Protocol (IP) [54] based networking, devices identify themselves and others with an IP address. An address carries information about where on the network the device is located, affecting how packets sent between devices should be routed. When an intermediate computer or router receives a packet, it will compare the destination address to entries in a routing table to make a decision about where to forward it.

In IP based networking, packets move independently from one another. Transmission Control Protocol (TCP) [55] is the most commonly used protocol to create an abstraction of a point-to-point connection on such a network. TCP is based on a concept of ports, and it ties address-port-pairs as the endpoints of a connection; a TCP connection is hence a quadruple consisting of two IP addresses and two ports. TCP works perfectly on a static network, but if a device suddenly moves from one network to another, the following things will occur.

The device will be assigned a new address that is usually not known in advance. As TCP does not have a secure mechanism for updating the endpoints of a connection, connections that were established using the previous addresses are disconnected and need to be re-established. Updating the endpoints using an insecure protocol would enable hijacking the connection. Most applications have no way to know if the new address really belongs to the same device; its requests need to be authenticated in some other way. During this process, some program state may be lost. Packets sent to the old address will not reach the intended device, and might even reach an incorrect device that has now been assigned the address. If the communication is unencrypted that device may be able to read it, and perhaps even respond to it.

One of the obvious characteristics of mobile devices is that they move. Users carry their devices with them wherever they go and connect them

to whatever network they have available when they need to connect to the Internet. On cellular networks, their address might stay the same when traveling from the vicinity of one base station to another because of handover functionality built into the network [4], but the same is not true when moving to other kinds of networks. The users might connect their devices to the campus Wireless Local Area Network (WLAN) when they're working, have the device automatically switch to mobile broadband when they step out of the door, and switch to WLAN again once they get home. The addresses of these devices change quite often, and it is impossible to reliably predict what the next address of the device will be.

The TCP/IP protocol stack is built of layers, which means there are many points at which to address the problem. The depicted problems are observed on the application layer because they are inherited from the transport layer that fails to accommodate dynamism from the network layer. Some programs, such as Mosh [66], address the problem on the application layer. Mosh itself can tolerate disconnections and address changes, but this doesn't help other applications.

Multipath TCP [2, 13] addresses the problem at the transport layer. Traffic can be spread across multiple paths, and as long as at least one path is available at all times, migration between networks should work. Multipath support for TCP would benefit the vast majority of applications, but not all communication is carried over TCP. Online games, for example, have been reported to suffer degraded network performance when using TCP, because it is too sophisticated for what the games require [6]. A separate mobility solution would hence be needed for other transport layer protocols as well.

Mobile IP [49] and Virtual Private Networking (VPN) address the issue at the network layer. In both solutions, the client can have a public IP address that always remains the same. Both solutions rely on a server, a gateway, or a *home agent* to hold a static public IP address, and forward the received packets over a tunnel to the target host. Typically the packets need to traverse triangularly through the gateway or home agent, despite more direct route being available.

Software Defined Networking (SDN) [11] and solutions such as Virtual Private LAN Service (VPLS) [27] attempt to solve the issue at link layer. They require sophisticated networking devices, such as programmable switches, to become more commonplace. One example is SMOG [21], which provides seamless VM migrations across data-centers without changes to software that is accessing or running on the VM. SMOG uses VPLS to span a single Layer 2 broadcast domain across multiple data-centers, and employs route advertisements to divert traffic to use the most suitable paths, in order to provide seamless migration of VMs across data-centers. While SMOG doesn't require changes to the applications, it only provides mobility to the server side, and migrations across Autonomous System (AS) borders are still an issue.

In this thesis, we focus on Host Identity Protocol (HIP) [42], which addresses the issue transparently to the applications, by adding a shim layer between the transport and network layers. The key idea is that the shim layer provides the upper layers a location-independent identifier while performing transparent address translation between the identifier and a routable, changeable address at the network layer. We discuss how HIP can support Cloudlets in computation offloading scenarios. One of the main the challenges in this work was that no working implementation of HIP existed that would run on current versions of Android. We chose to work on HIP for Linux (HIPL), because an earlier port for Android already existed, but which had to be modernized and rewritten to a large extent.

We have structured this thesis as follows. In section two, we discuss offloading and the use of surrogates in mobile computing in general, and different approaches to offloading. In section three, we take a deeper look at Cloudlets, starting with the general model, and then continuing with specifics about the Cloudlet prototype Elijah. In section four, we examine the HIP protocol in general. Then we discuss features that make it useful to be used with Cloudlets in section five. In section six, we discuss the process and changes of porting HIPL to the Android operating system, and, in section seven, we present benchmarks and results. Finally, we discuss some related work.

2 Mobile offloading

Mobile offloading, or cyber foraging [59], refers to transferring computational workloads from a mobile device to be executed on another nearby computing device, often called a surrogate. The motivation is typically to take advantage of its available resources, such as computing power or electricity. Figure 1 illustrates a generic division of responsibilities in offloading.

In this section, we examine characteristics and motivations related to mobile offloading, and take a look at some of the programming frameworks designed to support it.

2.1 Characteristics of mobile offloading

Porras *et al.* [53] divide cyber foraging into six steps: surrogate discovery, application partitioning, placement decision and cost assessment, trust establishment, task execution, and environment monitoring.

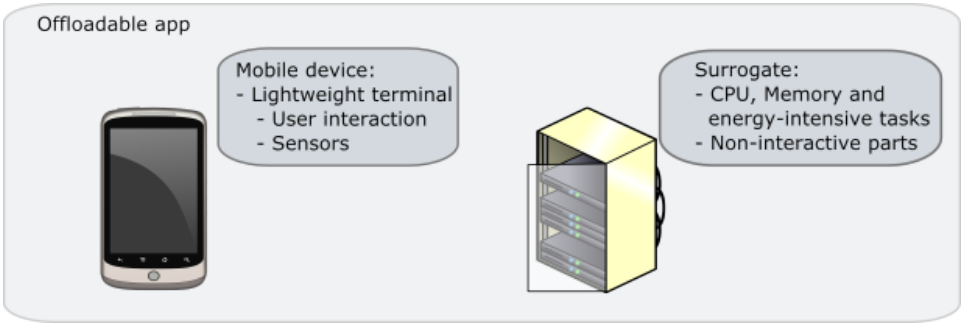

Figure 1: A mobile device and a surrogate.

Surrogate discovery is the process of finding nearby computing devices. This is usually done using a service discovery protocol. Many different kinds of service discovery protocols have been designed, with different properties. Some protocols rely on a central registry and are better suited for static environments. In others, the registry may be distributed, or the protocol may rely on the surrogates to broadcast advertisements of their services, or quietly listening to broadcasts and replying in unicast. Some protocols also combine these different approaches. The mobile devices employing discovery protocols may treat surrogate availability as a generic service similar to printing or web caching, or they may utilize details about the surrogates to find the best suited ones.

Application partitioning is the step of identifying which parts of the application can be offloaded. The application is partitioned into locally executable code and remotely executable code. The partitioning can be done manually by the programmer, or with the aid of a tool that helps with or even automates the process. Partitioning of the application is subject to certain restrictions. Generally, code that relies on local resources, such as code that reads input from a touchscreen, or otherwise interacts with the user, can not be offloaded because the input/output (I/O) devices it needs are at the mobile device, and not at the surrogate [53, 57].

Placement decision and cost assessment are processes of determining whether, and where, it makes sense to offload tasks that can be offloaded. The goal here is to make a plan for offloading. Offloading some processing to a surrogate imposes overhead on network and energy consumption. Before offloading makes sense, the benefit gained by offloading must be larger than the overhead. The planning step for offloading should take into account both the available resources of the mobile device and those of the surrogate, and the bandwidth and energy required to transfer the code, inputs and outputs. This step may also need to take into account the monetary cost of offloading and the user's priorities, should the surrogates be available as a commercial service.

Trust establishment means ensuring that the mobile device and the surrogate can trust each other. If both parties blindly trust each other, a malicious mobile device could consume all the surrogate's resources, thus causing a denial of service to other users, compromise other users' computations or subject the surrogate to other kinds of misuse. A malicious surrogate on the other hand could alter offloaded computations, illicitly reveal them to others, or hinder the tasks that the mobile user is trying to achieve by lying about its resources.

Task execution is the step where the actual offloading takes place. Once the application has been divided into locally and remotely executable parts, trusted surrogates are available, and the plan for offloading exists, the application needs to communicate the offloaded code and inputs to the surrogate, and transfer the results back.

Environment monitoring and application adaptability are the processes of an application adapting to changes in its environment. Whether it makes sense to offload something to a surrogate depends on the situation, which

can change over time even during the application's execution. If a surrogate suddenly becomes unresponsive, the application needs to cope with the changed situation and find a new surrogate, execute the code locally, or change the application's behavior.

2.2 Approaches to offloading

The offloading approach gravely affects the flexibility and resource utilization of an offloading system, and the best approach may depend on the application. The three most commonly used approaches are the RPC, the mobile code, and the Virtual Machine (VM) based approaches [53]. Table 1 summarizes the three common approaches.

In the Remote Procedure Call (RPC) based approach, a server component with implementations of the offloadable functions is running at the surrogate. The clients send their calls to those functions to the surrogate, and the surrogate executes the called functions and sends the results back. A purely RPC based approach doesn't support installing custom code from the client on demand. Since the surrogate will be executing only predetermined functions, their implementations can be precompiled and heavily optimized in advance.

Approach	Executes	Language for offloaded code
RPC	Predefined functions	Language agnostic
Mobile code	Arbitrary code written for platform's execution environment	Determined by framework, typically must run on the same VES
Virtual machine	Arbitrary code written for arbitrary execution environment	Can be language agnostic, but app or framework may restrict, VM format determined by framework

Table 1: Offloading approaches

In the mobile code based approach, the surrogate provides a generic execution environment, and the mobile client is allowed to send custom code to the surrogate to be compiled and run. This approach is used especially with code that runs in a Virtual Execution System (VES), such as code written in Java or the Microsoft .NET languages, because such languages are portable and architecture agnostic. Both the ARM architecture used in mobile devices, and the x86 or AMD64 architectures typically used in the surrogates can execute the same bytecode. Following this approach, a surrogate may also delegate tasks to other surrogates to further increase performance.

In the VM based approach, the remote execution environment is isolated in a virtual machine or container, and can be used to make any of the other offloading approaches available on demand. The surrogate may have a separate VM per application, or per user. The per-user VM may be altered, or even completely constructed by the individual user. The VM can host a different operating system than the surrogate, which makes this approach more flexible than the other approaches. On the other hand, the resource requirements for this approach are likely to exceed those of the other approaches, because resources need to be allocated for a complete virtual computer along with an operating system and the necessary applications. Containers, such as Docker[1], attempt to mitigate this requirement.

2.3 Motivations for mobile offloading

In the previous section, we discussed mobile offloading, and some different ways to approach it. In this section, we discuss some of the different applications.

Traditionally the resources of mobile devices are rather restricted, at least in comparison to off-the-shelf desktop or server computer hardware. The server machines, that are used as surrogates, can offer a more powerful processor and more memory to the applications than the mobile devices themselves can. This can speed up tasks that were previously run on the mobile device, and enable tasks that were previously not possible.

Another potential use case is offloading of network traffic. Running e.g., a BitTorrent client on the mobile device stresses the wireless network, and the performance may not be as good as that on a wired connection. A surrogate could be used to participate in the BitTorrent content dissipation and the mobile device can just download the downloaded files from the surrogate. Assuming that the surrogate itself is connected through a wired connection, this would cause less traffic on the wireless links, resulting in less congestion on the wireless links.

At locations where the Internet connection to the core network is slow or unreliable, a surrogate could be used to help with bulk file transfers. For example, backup software might benefit from the presence of a surrogate by being able to quickly transfer the backup over to it using a fast and reliable

[1]Docker: `http://docker.io`

WLAN link, and let the surrogate transfer the backup over to the remote cloud when the network allows. This would free the mobile device to be able to leave the network sooner without delaying the transmission.

Offloading can also be used to preserve battery life. Mobile devices are battery powered, whereas surrogates draw their power from the main electric outlets or possibly from green energy sources. Energy-wise, computation and networking are usually more expensive to the mobile device than they are to the surrogates. If the energy-cost of transferring the data over to the surrogate is smaller than that of performing the computation locally, the use of surrogates would result in a better battery life.

2.4 Offloading frameworks

A number of application development frameworks have been designed to support cyber foraging. They follow different architectural approaches, but most of them address all of the aforementioned tasks related to computational offloading. We next go through a number of the currently most popular cyber foraging frameworks.

Framework	Approach	Hardware platforms	Software platforms	Programmer involvement
MAUI	Mobile code	Any	CLR	Code annotation
Cuckoo	Mobile code	Any	Java VM	Separate implementations (optional)
Scavenger	Mobile code + RPC -hybrid	Any	Python	Separate implementation
ThinkAir	Virtual machine	x86	Java (Android)	Code annotations
CloneCloud	Mobile code	Any	Java VM	None
Cloudlet	Virtual machine	x86	KVM	Application design + overlay creation

Table 2: Comparison of offloading frameworks

MAUI [10] is a framework that follows the mobile code based approach and offloads computation dynamically, at run-time, to a MAUI server. The programmer can suggest parts of the program for offloading by annotating methods in the source code, and a component called MAUI Solver decides whether or not to actually offload the method. This design allows the software to run completely locally when a MAUI server is not present, and enable offloading when one becomes available. MAUI was designed to target the Microsoft Common Language Runtime (CLR), because the platforms for the server and the mobile device usually differ in the instruction set (x86 on server, ARM on mobile device), and CLR can run the same compiled byte code on both instruction sets.

Cuckoo [25] is an offloading framework targeted for the Android platform.

It takes advantage of the activity/service separation that is prevalent in Android application programming, where activities interact with the user and services perform the computational tasks. Cuckoo intercepts requests to services and opportunistically offloads the requests to a server. Like MAUI, Cuckoo follows the mobile code based approach, but the programming model for Cuckoo allows the methods to have different implementations for local and offloaded execution, aiming to a more efficient use of resources (e.g., increased parallelism) when offloaded. Since Android's Java code also runs in a VES, the difference in platforms should cause no problems.

Scavenger [34] is a cyber foraging framework that targets all mobile platforms. Its approach is something between the RPC based and mobile code based approaches. Scavenger transmits Python source code and input data as part of the RPC call, uses the surrogate to apply the code to the data, and returns what the code returned as the result.

ThinkAir [30] follows the VM based approach and runs "clones" of an x86-version of Android. The framework identifies the offloadable parts by the programmer's annotations in the source code. It also provides a customized Native Development Kit that is used to compile native libraries for both ARM- and x86-versions of Android. It supports an infrastructure that allows the clone to scale out into more clones in order to provide improved parallelism, and is able to cater to the needs of more than one user simultaneously.

Using the above mentioned frameworks, the programmer still needs special expertise on how to design their program to support offloading. SmartDiet [57] is a system designed by Saarinen *et al.* to help programmers with this task. SmartDiet analyses method call trees during program execution to find out which parts of the program consume most energy, to find where non-offloadable features have been used, and then to provide the programmer with advice on how to change the program to make as many parts offloadable as possible. It should be noted, that SmartDiet itself is a design aid utility and does not function as an offloading framework, but was considered relevant for this chapter.

CloneCloud [7] follows the mobile code approach also, and aims to reduce programmer involvement even further. It works with unmodified applications, analyzing them at run-time and opportunistically migrating threads to be executed on a server. CloneCloud relies on a modified version of Dalvik,

Android's Java Virtual Machine.

Cloudlets [60] follow the VM based approach. Before the offloading can start, the client provides the surrogate with an overlay image, which the surrogate will then combine with a base image to construct a completely customized virtual machine. This approach gives the programmer complete liberty to decide what will be run in the VM. The Cloudlet model in itself does not assist with designing the application, but it can in theory be used in combination with any of the aforementioned frameworks.

More thorough surveys on these and other computation offloading from mobile devices can be found in the works of Abolfazli *et al.* [1] and Kumar *et al.* [35]. In the next section we take a closer look at Cloudlets, and especially the Elijah Cloudlet system designed and developed at Carnegie Mellon University.

3 Virtual machine based Cloudlets with Elijah

Cloudlets [60] are a system for virtual machine-based cyber foraging.[2] The most notable prototypes, Kimberley [67] and the more recent Elijah[3], have been designed and implemented by researchers at Carnegie Mellon University (CMU). In this section we discuss Cloudlets, with a focus on the Elijah prototype. However, as Cloudlets are based on VM technology, we shall begin by discussing some background.

3.1 Virtual machines, cloning and migration

A Virtual Machine [15] is essentially a piece of state that represents a computer to a Virtual Machine Monitor (VMM), a program that manages virtual machines. The state of a VM is called a VM image that can be stored in a file like any other state from a computer program. Like any file, it can be copied or transferred over a network. A typical VM image contains information about the virtualized hardware, the contents of the VMs hard-drive, and, in case it's an image of a paused VM, also the contents of its main memory and the state of its processors.

Copying a VM image is called cloning. Moving the image to another host is called migration. Cloning can be used to make new VMs that share a common base. e.g., clones of an image with a clean, freshly installed operating system can be used to make a collection of different VMs that run the same operating system. In migration, the image on the original host is discarded, and the completely identical copy of it is resumed on the new host, essentially meaning that the VM has moved from one host to another.

The two common types of migration are called offline migration and live migration. In offline migration, the hypervisor pauses the VM, and sends the specifications of the VM to the hypervisor on the receiving host,

[2] Actually the term 'Cloudlet' is somewhat overloaded, even within the field of computer science. Objects representing application services in CloudSim are called Cloudlets [5]. S. Ibrahim *et al.* introduced a MapReduce-framework for virtual machines called Cloudlet [20]. Koukoumidis *et al.* suggest a caching scheme or cloud service design model called Pocket Cloudlets [31]. Tao Lin and Shuhui Wang propose Cloudlet-screen [39], a centralized multi-user computing system accessed through simple high-resolution terminals. Belalem *et al.* describe a cloud resource management model where Cloudlets are auctionable units of service [3]. Akamai holds a US trademark (USPTO #85027916) for the name Cloudlet and NephoScale uses it to offer on-demand virtual servers (`http://nephoscale.com/cloud-servers/`).

[3] `http://elijah.cs.cmu.edu/`

which reserves resources for the VM. It then sends the image of the VM to the receiving host. When the transfer is complete, the hypervisor at the receiving host resumes the VMs execution and the migration is complete. In some cases when the two hosts have access to shared storage, it is possible to skip the transmission of the hard-drive contents. During an offline migration, the VM is inaccessible.

In live migration [8], hypervisor does not freeze the original VM in the beginning, but instead lets it continue operating. The hypervisor sends the specifications of the VM to the receiving host, which reserves the required resources. Then the original host starts sending the full image of the VM over to the receiving host, and at the same time, starts tracking for changed parts ("*dirty pages*") in the VM's main memory and hard-drive. Once the complete image has been transferred for the first time, the original host starts a new tracking for dirty pages, and transfers the dirty pages from the previous iteration. Collecting and sending of dirty pages is reiterated until the number of dirty pages after an iteration is small enough, or until another condition, such as a time limit for attempting live migrations, is met. After the iterations, the original host freezes the VM, sends the remaining dirty pages and the CPU state, and the new host resumes the VM. In live migration, the VM remains accessible during the migration and its downtime can be kept very short. Should the migration fail, e.g., due to the receiving host not being able to resume the VM, it can still be kept running on the original host.

Migration requires support from the hypervisor. For migration to be successful, the abstraction of a computer provided to the VM must be similar enough on both hosts. Successful migration depends on how much actual hardware is exposed to the VM and how similar the host machines are. Virtualization platforms account for hardware heterogeneity by abstracting and hiding things such as special CPU instructions from the VMs. To the best of our knowledge, no hypervisor readily supports migrations to or from different hypervisors, but many do support importing hard-drive images from other hypervisors, allowing VMs to be started, but under a different set of virtual hardware [32, 61]. As an external solution, Liu *et al.* propose an inter-VMM shimming framework called Vagrant [40], which selectively reconstructs state created under one VMM in the format used by another VMM, and converts between the migration protocols used by different VMMs, en-

abling live migrations between heterogeneous VMMs.

Elijah's function can be thought of as a special case of offline VM migration. The base image is created on one host, and the paused full state is copied to any host that acts as a Cloudlet server. The default configuration in Elijah presents the VMs with a virtual processor called "*qemu64*". This processor is one of QEMU's standard options and is aimed as the greatest common nominator amongst the x86-64/AMD64 family of processors, i.e. to expose the largest common set of processor instructions found in such processors. A Cloudlet VM should, hence, be migratable across different hosts, whose processor is compatible with this virtual processor.

3.2 The Cloudlet model

The Cloudlet model consists of a Cloudlet host, a mobile device, Cloudlet Virtual Machines and possibly a remote cloud. Figure 2 depicts the structure of a general Cloudlet infrastructure.

The Cloudlet hosts are commercial off-the-shelf (COTS) servers that are powerful and network-wise in close proximity of a network access point, such as a wireless hotspot. They are equipped with hypervisor software for running virtual machines, and Cloudlet management software to manage the VM images, the synthesis process, interacting with mobile clients etc., and possibly a collection of Cloudlet base images. The Cloudlet management software can also advertise its presence so that clients can find it using a service discovery mechanism.

The Cloudlet host provides its services to Cloudlet-aware software running on mobile devices, such as smartphones or laptop computers. As the Cloudlet system is still experimental, actual client software exists yet only for Linux and Android operating systems for the time being.

The mobile device accesses the resources of the Cloudlet host by starting an instance of a customized virtual machine, a Cloudlet VM, and by connecting to it from some Cloudlet-aware application. The Cloudlet VMs are tailored for each application. The image of the VM is constructed from a shared *base image*, which can be standardized and typically comprises most of the image, by applying an *overlay image* provided by the application. In Cloudlet terminology, this construction process is called VM synthesis. The base-overlay separation serves to answer one of the classic challenges of Transient PCs, e.g., how to handle large parcels efficiently [62]. To further

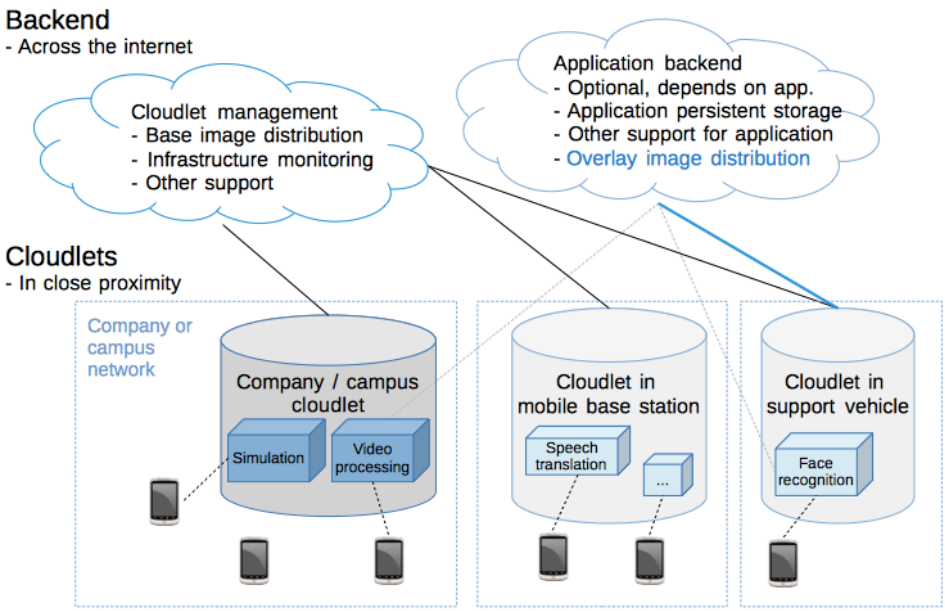

Figure 2: Overview of an example Cloudlet infrastructure.

reduce the size of the overlays, Ha *et al.* have implemented a number of optimizations [18].

Finally, a remote cloud can exist in the background. The remote cloud can be used as a Content Delivery Network (CDN) for distributing base-images and overlay-images, for infrastructure management, to provide persistent storage or other kinds of support.

While computation can be offloaded into a remote cloud, performing computation in a local Cloudlet does provide some benefits. Cloudlets can be thought of as extensions for clouds that bring the cloud closer to the end-user.

They should be placed on the local network, ideally only a few hops away from the network access point. Being located in the local network offers lower latency and higher bandwidth than would be achievable over the Internet. Low latency especially has been shown to have an essential role when serving interactive applications [9].

Since Cloudlets reside on the local network, they can even be accessible when the Internet is not. One of the things this implies is that the computational resources are usable in field conditions, e.g., when the Cloudlet is placed on a locally connected support vehicle, or in situations where global

connectivity is missing or unreliable [63].

3.3 Elijah

Elijah is a prototype implementation of a Cloudlet system. It consists of several pieces of software for Linux, and a client application for Android. In this section, we walk through the different components making up Elijah.

3.3.1 Linux applications

Elijah consists of a customized version of the QEMU virtualization software, software for creating the VM images, client and server software for VM synthesis, an extension for OpenStack[4] to integrate Elijah with it, and a service discoverability tool based on Avahi[5].

3.3.2 Android client

The Android application for Elijah is a simple client to interact with the Cloudlet synthesis server. It supports the simple synthesis server, and also the OpenStack integration. Location of the Cloudlet server can be discovered through Avahi, or entered manually. The application can send an overlay image to the synthesis server and shut down the synthesized VM when it's no longer needed.

3.4 Structure of a Cloudlet VM

A Cloudlet virtual machine image is constructed of two parts: a base image, and an overlay image. The purpose of the base image is to set a generic foundation, which is later refined into an application-specific image by applying an overlay image.

The base image consists of a hard-drive image containing an operating system and common libraries, an image of the main memory, and the CPU state, where the operating system is freshly started. The base image usually accounts for most of the space consumed by the full image. Ideally the base images would be standardized and shared among as many overlay-images as possible to keep the number of existing base-images low. It should

[4] `http://www.openstack.org`
[5] `http://avahi.org`

be noted, that the same base image can be shared by any number of overlay images.

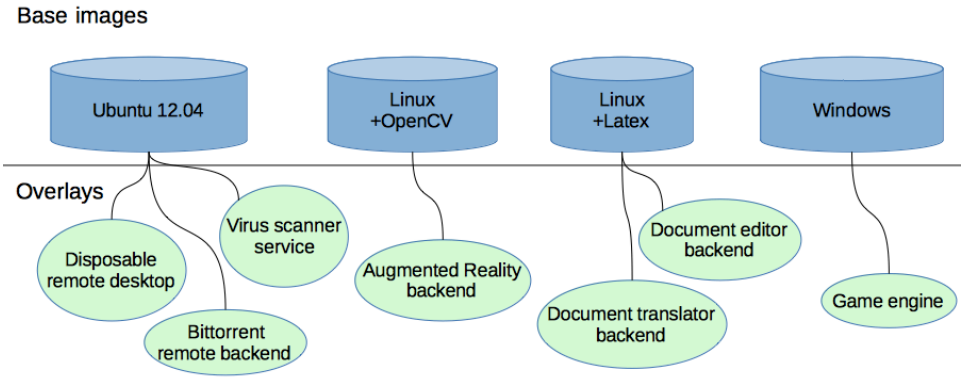

Figure 3: Base- and overlay-images.

The overlay image refines the base image and contains the application-specific part. It also consists of a hard-drive image, a main memory image, and CPU state, or more correctly their differences (or *delta*) to the ones in the base image. It contains the information on how the contents of the main memory and hard-drive have changed, starting from the base image, ending in the application having been installed and started. When resuming a Cloudlet VM, this delta is applied to the base image to reconstruct the full state. Figure 3 depicts the division between base images and overlay images in Cloudlets.

3.5 Image creation and virtual machine synthesis

In the previous section, we studied the structure of a Cloudlet VM. In this section, we take a look at how a Cloudlet VM can be created and how to run it.

3.5.1 Creating the images

The creation of both the base and overlay images is a relatively straightforward process. For the base image, start by creating a QEMU raw VM hard-drive image, and install an operating system on it. Install also any libraries that should be available to the overlays. This part can be done without invoking the Elijah software.

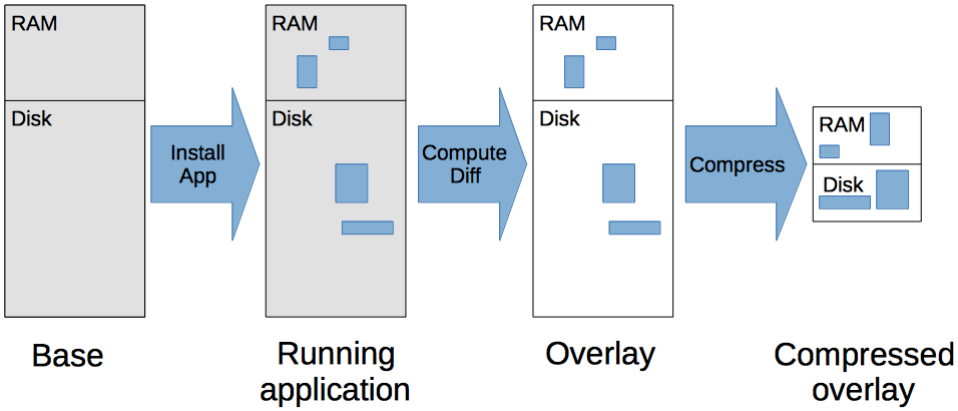

Figure 4: Creation of an overlay image.

When the installation is ready, start the VM in Elijah in the base image creation mode. This will open a Virtual Network Computing (VNC) remote desktop window where you can configure the VM into exactly the desired state. When ready, close the VNC window. This causes Elijah to freeze the VM, and store the full state (hard-drive, main memory, processor state) of the VM into a file. This file is called the base image.

For the overlay image, start by launching a virtual machine from the base image in Elijah's overlay creation mode. Once the Elijah displays the VNC window with the base image, use it to install and launch the desired application. Once the application is ready, close the VNC window again. Elijah freezes the virtual machine and computes the binary difference (or *delta*) between the full states of the base image and the frozen VM, and stores the delta in a file. This file is called an overlay image, and it can later be used together with the base image to reconstruct the frozen VM. Figure 4 illustrates the creation of an overlay image.

3.5.2 VM Synthesis

When the Cloudlet VM exists as the base image and the overlay image, they can be combined to reconstruct the full VM image. The overlay image contains the information which base image it was derived from. Elijah uses this base image as the starting point, decompresses the overlay image and applies it over the base image as binary patch, resulting in the full VM image. It then resumes the reconstructed VM, which is now in the state in

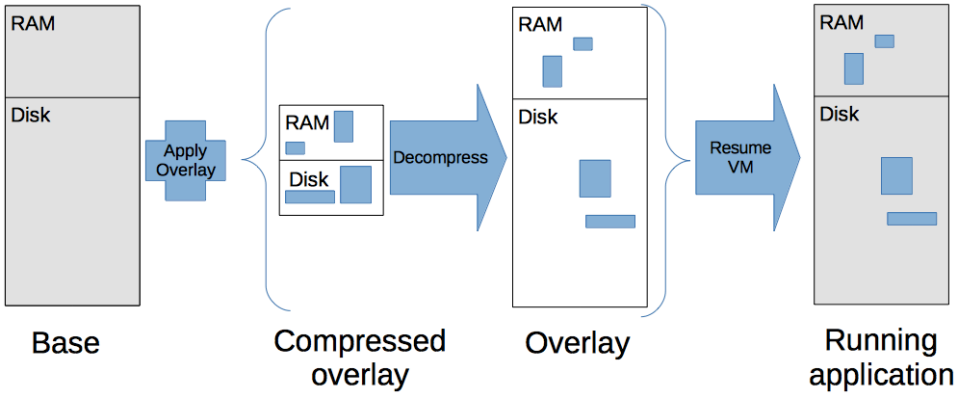

Figure 5: Synthesis of a Cloudlet VM.

which it was at the end of the overlay creation, typically with the desired application installed, launched and ready to serve the mobile application. Figure 5 represents applying the overlay over the base image.

The larger the overlay is, the longer it takes to transfer it from the mobile device, and also the longer it takes to apply it on the base image. To minimize the size of the overlay image, keep the *changes* from the base image as small as possible. Removing unneeded parts of the base image in the overlay creation step may result in a larger overlay image, because removing parts makes the image more different from the base image i.e., bigger delta.

3.6 Image distribution

In Elijah, base images are stored on Cloudlet hosts, and overlay images are stored in mobile devices or in cloud. To synthesize a Cloudlet VM from the images, the mobile device sends the application specific overlay image to the Cloudlet. Alternatively, the mobile device can send a URL where the overlay image can be downloaded, which reduces the work of the mobile device if the Cloudlet has access to the Internet. Pointing the Universal Resource Locator (URL) to a CDN with a good global presence can greatly improve the performance of this approach. The Cloudlet can also cache the most recently used overlay images, or pre-download images that are expected to become popular. The mobile devices would not need to transfer such images separately each time they are used.

Referring to the overlay image using a URL provides dynamism. The

URL can be pointed to a location that always has the latest version of the overlay image. This would make it possible to always use the latest image when Internet access is available, and fall back to the image on the mobile device when Internet is unavailable. It can also be used for content that is tailored for the current location, where local support data e.g., for an Augmented Reality (AR) application can be distributed in the overlay image.

3.7 Applications

Cloudlets provide a customizable software environment in which to run virtually any kinds of applications. As a Cloudlet VM is technically a complete computer, it can be used to host any of the other frameworks discussed in section 2.4. The traditional examples of offloadable applications remain relevant with Cloudlets as well. Application examples discussed in literature include speech and facial recognition [52, 17], live speech translation [17], object recognition, body language interpretation and Augmented Reality [18].

In addition to the traditional cases, the VM-based approach of Cloudlets allows a kind of portability use case as well. The offloaded code is written for the hardware platform of the Cloudlet host, instead of the client. This can be used, e.g., to play computer games that were never actually made for the mobile device [24].

3.8 Mobility challenges

Building a complete mobility solution for Cloudlets remains outside the scope of this thesis, but we find it a meaningful subject to discuss nonetheless. In this section we identify challenges related to building a complete mobility solution, and discuss more deeply the subset that we aim to solve.

3.8.1 The complete mobility solution

Building a complete mobility solution for Cloudlets would require solving at least the following challenges:

- Roaming
- Migration
- Host discovery
- Resource management
- Application lifecycle / context management
- Accounting
- Federation

By roaming, we mean the capability to maintain connections while one of the parties, be it the client, the Cloudlet host, or the VM, is changing networks. The solution should also be secure in the sense that outside parties cannot hijack the connection. To this particular challenge, we propose HIP as the solution.

By migration, we mean the capability of the hosting platform to perform live migrations on the Cloudlet VM. The customized version of QEMU[6] used in Elijah did not support live migration of the VM. Should a Cloudlet VM move by means other than re-instantiation, working support for migration is needed.

Host discovery is the challenge of finding Cloudlet hosts, and by this we mean finding viable targets for migration. They may or may not reside on exactly the same network as the client, but a host found nearby may still be better than the currently connected one that has become too distant due to the client movement.

Resource management is the challenge of finding and enforcing policies on how to allocate the resources on Cloudlet hosts. Context management is the challenge of dealing with application, client, and contract based matters. A complete mobility management system would use information from resource and context management systems to decide where to place the Cloudlet VMs and whether, where, and when to migrate them.

Accounting deals with whoever uses the Cloudlet hosts, that how many resources they can use and when can they use them. Accounting may also

[6] `https://github.com/cmusatyalab/elijah-qemu`

deal with how many resources a specific client is allowed to use, and with prioritization between different users or groups. It should be noted, that in commercial use, accounting is typically also linked to billing.

We believe that a complete mobility management solution should also take into account that the best Cloudlet host for the client could sometimes belong to another operator. Infrastructure that enables the client of one Cloudlet provider to use resources provided by other providers can benefit all parties involved. Hosting of a Cloudlet infrastructure is analogous to hosting Infrastructure as a Service (IaaS). Kim *et al.* have shown that in cloud computing, small IaaS providers are a special group that can especially benefit from forming a federation [26]. Cloud brokerage and federation solutions, e.g., the ones discussed by Fowley [14] have solved many of the challenges related to this domain. From the Cloudlet perspective, we believe that all of the above challenges should be solved before a truly acceptable solution can be found to this one.

3.8.2 The roaming challenge

To us, an adequate roaming solution for Cloudlets should:

- Allow parties to resume the connection without disconnection.
- Allow parties to resume the connection without undue delay.
- Preserve the connections' low delays and high bandwidths.
- Prevent others from hijacking connections.
- Preserve confidentiality and integrity when moving to hostile networks.
- Not depend on the Internet.

When a device roams from one network to another, the mobility solution should make sure that any state coupled with the connection is preserved, so that roaming is as seamless as possible. The delay involved with resuming the connection should at least be short enough, so that the user will not start looking for problems in their connection.

The roaming solution should not unreasonably degrade the performance of available connections. Some of the key attributes in Cloudlets are reliable and fast network connections between the user and the Cloudlet[7]. A roaming solution that would sacrifice key design points of the basic solution would obviously be unacceptable.

[7]Key attributes listed in: `http://elijah.cs.cmu.edu/`

The mobile clients, the Cloudlet hosts, and the Cloudlet VMs should be able to tell each other if they have moved. An outsider, however, should not. A roaming protocol must be secure in a way that any claims of roaming to other networks are not trusted before they are securely verified. Also, a good roaming solution does not make the communications more susceptible to eavesdropping even if the device is forced to switch to a network that offers less safety.

Finally, Cloudlets are designed to work as long as the client has local connectivity to the Cloudlet [63]. A roaming solution that would require external connectivity would also violate an important design point and should be deemed unacceptable.

In the next chapter we have a look at one solution to the roaming challenge, called Host Identity Protocol.

4 Host Identity Protocol (HIP)

Host Identity Protocol [42, 46] is a network layer (L3) overlay protocol for creating secure end-to-end connections between two hosts. Hosts are identified by self-generated cryptographic identities called Host Identities (HIs). The identities are referred to by specially formatted IPv6 addresses called Host Identity Tags (HIT), which can be used directly or through local IPv4 mappings called Local Scope Identifier (LSI). HITs are globally statistically unique IPv6 special purpose addresses[8] that belong to the Overlay Routable Cryptographic Hash Identifier (ORCHID) [45] address space.

The protocol is used to enhance security, and to support client-side and server-side mobility.

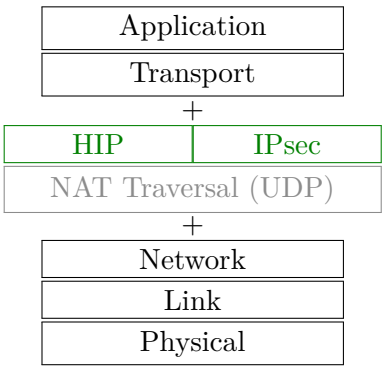

Figure 6: Protocol stack with HIP and basic NAT traversal encapsulation.

4.1 Separation of identity and location

Traditionally the addresses in Internet Protocol (IP) have served both as locators and as identifiers. The source and destination computers are identified by the IP addresses assigned to them. This has an inherent problem that should a device be assigned a new address, its identity, from the protocol's point of view, will also change. Domain Name System (DNS) provides a kind of identity/address split, in the sense that domain name records can be updated. When a host tries to connect with a host by name, it performs a lookup for mapping the name to an IP address. That IP address is then tied as a permanent endpoint for the connection and cannot be updated.

[8] `http://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml`

An update to a domain name record is effective for lookups that hosts make after the old record has expired from the caches of the DNS servers that are used.

Host Identity Protocol (HIP) adds an additional shim layer between the transport and network layers, as shown in Figure 6. An application that wants to communicate with another one identifies the destination host by its Host Identity Tag (HIT) or the corresponding LSI mapping. The HIP layer then needs to know a routable address where the target host can be reached, and it negotiates an end-to-end IPsec tunnel. The HIP layer translates HITs and LSIs into routable locators and vice versa. The HIP DNS Extension [44] supports attaching names to identities and locators. The HITs or LSIs cannot be updated either, but their respective locators can.

Many other protocols provide indirection through splitting the roles of IP addresses. Komu *et al.* conducted a survey of such protocols [28].

4.2 Control plane

HIP is structurally divided into a *control plane* and a *data plane*. The control plane is responsible for negotiating Security Associations (SAs), accommodating locator changes, negotiating details about cipher suites and related to HIP association state and IPsec.

4.2.1 Base exchange

RFC 7401 [41] specifies the details of the Host Identity Protocol. A HIP association is formed using a four-way handshake between an Initiator (*a client*) and a Responder (*a server*). The messages exchanged as part of the handshake are labeled I1, R1, I2 and R2. The Initiator sends the I-messages, to which the Responder responds with the R-messages respectively. Figure 7 shows the essentials of the exchanged messages and state transitions of the two hosts.

All the messages carry a numeric code for their type, the HITs for the source and destination hosts, and a HIP envelope. The I1 message is used for establishing initial contact. Its header carries the identifier of the message type, the HIT of the initiator, and the HIT of the responder, and the HIP envelope is empty. An all-zeroes value for the responder's HIT is treated

as an opportunistic initiation, and any HIP-aware host that receives the message can respond to it.

The R1 message carries the responder's HIT as the source, and the initiators HIT as the destination. Unless opportunistic mode was used, the HITs must match those in the I1 message. If the initiator or the responder uses an anonymous identity, the corresponding control bit must be set in the header. An anonymous identity is one that is regenerated periodically to thwart attempts on tracking of HIs. The HIP envelope of the R1 message carries a computational puzzle, along with a generation counter, parameters for Diffie-Hellman key agreement, details about supported encryption and integrity algorithms, the host's full identity and optionally a signed or unsigned nonce. It should be noted, that a new set of puzzles is generated periodically. The generation counter specifies the generation to which this specific puzzle belongs. As the R1 message reveals the identity of the host, the responder may instead choose to respond with an I1 message, effectively switching places with the initiator [41, page 19]. If both hosts have a policy to reply with an I1 message instead of an R1, the handshake between them cannot complete. R1 messages and solutions to the puzzles can be precomputed, which makes responding to I1 messages computationally inexpensive.

The envelope in the I2 message carries the solution to the computational puzzle presented in the R1 message, the initiators parameters for Diffie-Hellman key agreement, and the chosen encryption and integrity algorithm to be used for the Host Association (HA) and SA. The identity of the initiator is also part of the I2 message, but it may be encrypted using the chosen encryption algorithm, using keying material derived from the Diffie-Hellman parameters. The signed nonce is included in the I2 message. The initiator computes a Keyed-Hash Message Authentication Code (HMAC) [33] over the envelope, signs the envelope, and if the R1 message contained an unsigned nonce, appends it after the signature. Verifying the correctness of the I2 package is also computationally relatively inexpensive. The responder will only proceed with establishing the association if the I2 was verified successfully.

The envelope in the R2 message contains only a HMAC over the envelope, and the responder's signature. After successfully validating this message, the initiator considers the HA established. When application data starts flowing through the connection, the responder will also consider it established.

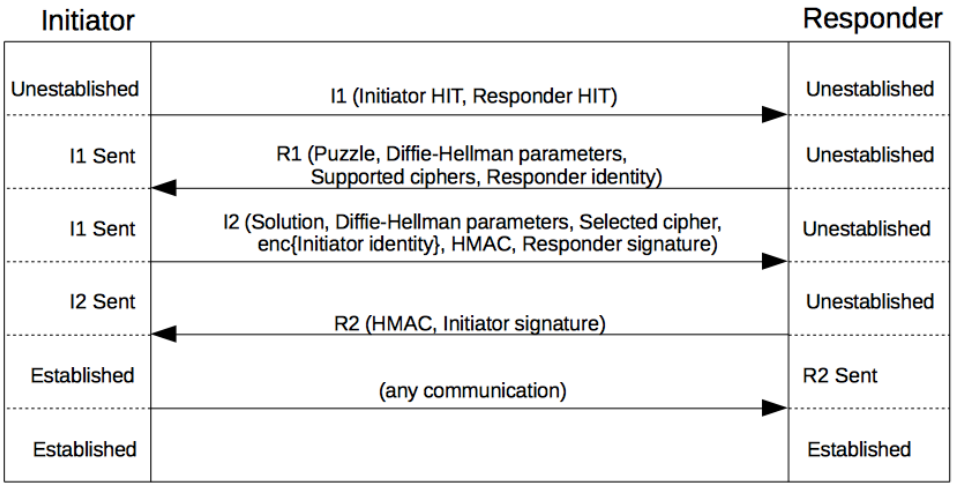

Figure 7: HIP base exchange

4.2.2 Update mechanism

A HA may need to be updated from time to time. Typically an update is needed when a host is relocated in a network, or when the session key used for encryption needs to be refreshed. HIP provides a three-step mechanism for updating the SAs.

When the locator of a HIP host changes, it sends a signed UPDATE message to the hosts to which it was connected. When a host receives an UPDATE message, it performs a so called return routability test, i.e., it verifies the update by sending back an echo request. If it then receives a valid response for the echo request, the host updates the locator in the corresponding SA. The update may optionally also contain re-keying material for the HA, initiated by either party.

HIP provides its services at the network layer, and is agnostic of any roles for applications. The mobility features work the same way on both clients and servers, meaning that HIP supports both client-side and server-side mobility.

4.2.3 Rendezvous

The HIP update mechanism works only when one of the two associated hosts moves at a time. Should both hosts move at the same time, they would both end up sending update messages to addresses that no longer

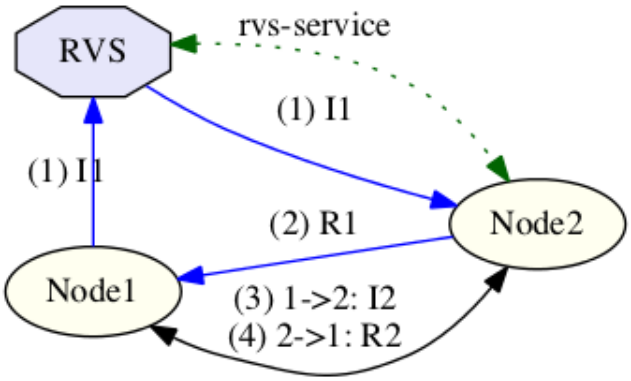

Figure 8: HIP handshake using a Rendezvous server.

reach the correct host. To accommodate mobility for both hosts moving at the same time, HIP Rendezvous (RVS) extension [38] exists.

The Rendezvous server relies on a stationary host that accepts requests to relay the I1-messages used in the HIP handshake, and the UPDATE messages used in the update mechanism. When a host moves, it updates its locator with the Rendezvous server. When another host notices that its messages are no longer reaching their destination, it reverts back to sending an UPDATE message through the Rendezvous server. By this time, the Rendezvous server should have the new locator for the target host. It forwards the UPDATE message to the target host.

4.3 Data plane

The *data plane* in HIP is responsible for encrypting and encapsulating traffic and transmitting it to the correct destination based on the SA set by the control plane.

The *control plane* of HIP supports negotiation of any encapsulation for the *data plane*. By default it uses IPsec in the Bound End-to-End Tunnel (BEET) mode for Encapsulating Security Payload (ESP) [23]. BEET mode is a lighter, but also a more restricted type of tunneling than the regular tunnel mode. The data address associated with plane can be dynamically reconfigured by the control plane, allowing for seamless handovers when network topology changes.

When a host sends a packet over a tunnel-mode IPsec tunnel, IPsec adds an additional IP header (*outer header*) before the original header (*inner header*), to identify the endpoints of the tunnel on a regular IP network. When the packet reaches the other endpoint of the tunnel, the added header is removed and the packet is routed to its destination using the original header. The link's Maximum Transmission Unit (MTU) limits the size of packets sent over the link, and each additional header consumes part of the MTU. Increased size of headers and decreased space for payload means more overhead.

In BEET mode, each endpoint stores the inner and outer addresses of the endpoints related to the SA. When a host sends a packet over a BEET-mode IPsec tunnel, instead of adding a second header, IPsec substitutes the original header with the outer header and sends the packet with only that single header. When the packet arrives at the other end of the BEET tunnel, IPsec maps it to a SA by its Security Parameter Index (SPI). IPSec then reconstructs the original header using information attached to the SA, substitutes the received header with the reconstructed inner header, and routes the packet to its destination. As the inner addresses are not communicated along each packet, this model introduces less overhead. Since the only factor mapping the packet to an inner address is the SPI of the tunnel itself, one tunnel can only support one pair of network-layer endpoints [23].

As long as the hosts at the ends of the tunnel understand the addresses in the inner headers, they do not need to make sense for other middleboxes in the public IP network. HIP uses HITs as the inner addresses in the tunnels. As the applications need to know only about the inner addresses, the outer addresses can be changed without affecting the applications' view of the connection. As HIP SAs are designed to be end-to-end, the restrictions imposed by BEET mode are acceptable, so HIP has the benefit of changeable locators and smaller overhead.

4.4 Summary of security

We shall approach the security of HIP via the classic CIA-triad: Confidentiality, Integrity, and Availability.

HIP protects confidentiality and integrity through the use of a secure key exchange protocol, encryption, signatures, and checksums. Key material for session keys is generated using a Diffie-Hellman key exchange. Because the

key material is complete by the time that the I2 message is sent, the identity of the other party to the communication can be concealed. HMAC codes are used to quickly detect a message may have been altered on the way, and signatures are used to verify the source and the integrity of messages.

HIP protects availability by including a puzzle as part of the 4-way handshake. The answers to the puzzles can be precomputed at the responder so that checking the correctness of an answer is computationally inexpensive. With precomputed answers, the responder does not need to reserve resources for establishing a state until after successfully validating that the answer to a puzzle is correct. Attacking a host with a stream of I1 and bogus I2 messages causes the responder to effectively ignore the initiator, consuming a very small amount of resources. Denial of Service (DoS) attacks, where the attacker does solve the puzzles, are also impractical because the work to compute the answer can be orders of magnitude larger than the work to verify it. Komu *et al.* evaluate the effectiveness of the HIP puzzle mechanism as a spam mitigation method for email servers, and conclude, that forcing a sender to solve a new puzzle each time that the server identifies a message from the sender as spam efficiently lowers the load on the server [29].

4.5 Network Address Translation

Network Address Translation (NAT) systems introduce a whole new set of challenges. Typically, a NAT device establishes a return route for responses to outgoing packets. Whether the responses must come from the same IP address or not depends on the NAT implementation. Nevertheless, this usually means that a device behind a NAT device can initiate connections, but being able to respond to initiations requires special configuration from the NAT device.

Assuming an established HIP association, if one of the hosts moves to a network that is behind a NAT device, it should be able to recover the connection using the normal update mechanism. If the other host also moves behind a NAT device, the successful recovery of the connection depends on the how the NAT device with the first host is configured.

Since the RVS-service is client-initiated, hosts that are behind NAT devices can be reached through their RVS-server locators. However, if the initiator is behind a NAT device, sending an I1 message to the RVS-server causes the NAT device to initiate a return path from the RVS-server instead

of the destination host. Since the original host will send its response directly to the initiator's locator from its own address, their NAT device may refuse to deliver it.

The RVS server may offer a relay service in which also the data-plane travels through the RVS server, allowing the connection to succeed even if both devices are behind NAT devices. This may have an impact on the connection's performance. There are also other NAT traversal solutions that may be used with HIP, such as Teredo [19, 65] and *Hi*3 [16] (derived from *i*3 [64]). Microsoft Windows, starting from Windows Vista, comes with an integrated Teredo client[9], and a client called Miredo[10] is available for Linux and BSD operating systems.

This has relevance to our work in providing a HIP implementation for mobile devices, in that mobile network operators are in the process of moving their consumer-grade Internet connections behind large NAT systems, so called Carrier Grade NATs [22]. Should the users of such access technology be allowed to communicate directly to each other, there is a demand for a proper NAT traversal solution that would support HIP.

4.6 Multihoming

A host that is connected to multiple networks is said to be multi-homed. Such a host usually has at least one address per network. It is possible for a HIP node to communicate multiple IP addresses to its peer during the base exchange or update. From the viewpoint of HIP, this means that multiple locators can be associated with one identifier. The same identifier can be used to address the host from any of its locators.

The benefits of associating multiple locators for an identifier are fault tolerance and the capability to balance load. If the host becomes unavailable through one of its addresses, the hosts which do know about the other locators can try reaching the host through the alternative locators. The different locators for a host would typically come from a different network. Addressing the host using the same identifier, but different locators balances the traffic between the networks.

It should be noted that HIP for Linux (HIPL) provides an extension

[9]Teredo in Windows: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa965905(v=vs.85).aspx`

[10]Miredo: `http://www.remlab.net/miredo/`

called *shotgun mode*[11] that causes the initiating host to send the I1 or update message to all known locators of a host when initiating a handshake or an update.

4.7 HIP Implementations

Several implementations of the HIP protocol have existed over the years. In this section, we take a look at some of the implementations that are currently available. We found evidence of other implementations, such as HIP for BSD (HIP4inter.net) and PyHIP as well, but at the time of writing this thesis they did not seem to be publicly available.

4.7.1 OpenHIP

OpenHIP[12] started as the Boeing HIP implementation, but it was renamed in 2006. The latest release (OpenHIP 0.9) supports Linux, 32bit versions of Windows, and OS X Tiger (10.4.6 - 10.4.10). OpenHIP consists of a daemon called *hip* and a set of utilities for managing it.

The *hip* daemon is responsible for everything related to running the protocol. It supports RVS, broadcasts to Local Area Network (LAN), and a DNS extension using the Top-Level Domain (TLD) *.hip* to specifically request HITs and LSIs. The *hip* daemon reads a configuration file called *known_host_identities.xml* to learn about other hosts that support HIP, and allows communicating with them using HITs or LSIs.

On Linux, previous versions of OpenHIP secured the traffic toward a target host also when referenced by its regular IP address, if the daemon was running in kernel mode on a patched kernel. The kernel mode support was dropped in OpenHIP 0.9.

hitgen is used during OpenHIP's installation process to generate a HI for the host. It can be given a set of parameters such as the desired algorithm and key length. *hitgen* is also used to generate the sample configuration file for *hip*.

hipmon is a simple OpenHIP management utility for Windows. It provides an icon to the system tray that can be used as a shortcut to start and stop *hip*, and to access logs and configuration files.

[11] HIPL Manual: `http://infrahip.hiit.fi/hipl/manual/ch09s02.html`
[12] OpenHIP: `http://www.openhip.org/`

4.7.2 HIP for Linux

HIPL[13] is an implementation of the HIP protocol for the Linux operating system. It was originally created as part of the InfraHIP project. HIPL consists of three parts: *hipd*, *hipfw* and *hipdnsproxy*.

hipd is a daemon that enables the host to communicate using the HIP protocol. It provides basic features such as the base exchanges, the update mechanism and manages IPsec. The hipd also supports the RVS and relay services, keeping idle connections alive using so called heartbeat extension, and certain locator discovery features such as DNS updates and broadcasts to LAN. The hipd supports addressing of hosts by both HITs and LSIs, but communication with LSIs requires hipfw.

The *hipfw* daemon provides basic firewall services around HIP, such as filtering traffic based on HITs, packet types, or state in a protocol run. hipfw also provides address translation and packet forwarding between LSIs and HITs. Using LSIs for communication in HIPL requires hipfw to be running with LSI support enabled.

hipdnsproxy places itself as the handler of DNS queries for the host, forwarding queries and responses normally, but intercepting responses that contain Host Identities. When hipdnsproxy encounters a DNS response that contains a HI, it will insert a mapping between the HI and the received locator to *hipd* and returns the HIT or the associated LSI to the requester. Using DNS for transferring HIs reduces effort of setting up SAs but is subject to the same vulnerabilities as DNS in general.

4.7.3 CuteHIP

CuteHIP [37][14] is an implementation of HIP by Dmitriy Kuptsov. It was created as a part of developing a rapid protocol development and experimentation framework for Java as the initial use case for the framework. CuteHIP is written mostly in Java, but uses native libraries for interacting with raw sockets and TUN/TAP-devices. A version of the native libraries is provided for Linux, OS X and Windows, but, in their paper about CuteHIP [37], the author only experiments on the Linux platform.

[13] Infrastructure for HIP project: `http://infrahip.net`
[14] `https://code.google.com/p/cutehip/`

5 HIP as a mobility solution for Cloudlets

Cloudlets are designed for augmenting mobile devices, and mobile devices tend to move. The usefulness of the augmenting solution can be increased if the user is allowed to move even during an ongoing session. Traditional networking is not adequate for this goal, and must be augmented with a mobility management solution. In this chapter, we view some of the mobility solutions available today, motivate our choice of HIP, and discuss the application of HIP to Cloudlets.

5.1 Mobility solutions

There are several protocols that can accommodate Cloudlet mobility management. In HIP [42], hosts are assigned (statistically) universally unique identifiers, which are mapped to locators that can be changed on demand even during connections. Hosts inform other connected hosts about their new locators directly, and non-connected hosts can find the new locators through a rendezvous service if one is used. HIP is an application agnostic solution, and supports mobility on all sides of the communication. In addition, DNS can be used to deliver Host Identities.

Protocol	Client-side mobility	Server-side mobility	Route optimisations	Android software
HIP	Yes	Yes	Yes	HIPL* (experimental)
Mobile IPv4	Yes	Yes	Proposed [51]	N/A
Mobile IPv6	Yes	Yes	Yes	UMIP*
IPOP	Yes	Yes	Yes	SocialVPN (experimental)
OpenVPN	Yes	Yes	No	Yes

Table 3: Comparison of mobility protocols. *) Needs custom kernel.

In Mobile IP [49], a host that is outside its home network is associated with two addresses: a *home address* and a *care-of address*. The home address is the host's permanent address in its home network, and the one that other hosts should use when contacting the host. The care-of address is the address assigned to the host by the network that it's visiting. When the host is visiting another network, the home agent receives any packets sent to the home address, and tunnels them to the host in its care-of address. The host responds to packets ether by sending them directly, while using its home address as the source address, or by reverse tunneling the responses

through the home agent. The first approach constitutes a route optimization technique called triangular routing, and will not work on networks that reject spoofed source addresses in outgoing packets. Mobile IPv4 is interoperable with standard IPv4.

Mobile IPv6 [50] is a related protocol that provides mobility for IPv6 hosts. Mobile IPv6 can work in two modes: *bidirectional tunneling* mode, and *route optimization* mode. In bidirectional tunneling mode, Mobile IPv6 tunnels all traffic between two hosts, in both directions, through the home agent. In route optimization mode, a host registers its care-of-address with its peers that support route optimizations, and the hosts communicate using their care-of-addresses directly. The bidirectional tunneling mode is compatible with IPv6 enabled hosts that do not support Mobile IPv6, and is also used where the route optimizations are unimplemented or prohibited.

An OpenVPN[15] server can be used to provide a tunneled virtual L2 bridge or a routed IP subnet, where the server acts as a central point, i.e., as a gateway or relay between different hosts, which can act as clients or servers, and potentially also as a gateway to the Internet. Addresses inside such tunnels are called internal addresses and they can be assigned statically. Hosts connected to an OpenVPN network with static addressing can always address each other using the same internal address regardless of their physical address. OpenVPN, can route all or part of traffic from a host to the virtual network, and such traffic always travels through the gateway. An implicit limitation is that clients that need to access a server using its static inner address need to have access to the virtual network produced by OpenVPN. While this may be feasible for in-house solutions, it may be less feasible to offer to the general public.

IPOP [12][16] is a new user-friendly system and protocol for constructing Virtual Private Networking (VPN) connections for pairs or groups of devices. IPOP uses the Extensible Messaging and Presence Protocol (XMPP) [58] in determining which devices should be connected in the virtual networks, and automates the task of forming secure tunnels between them. IPOP works in two modes: *SocialVPN* and *GroupVPN*. In SocialVPN, pairings are determined through friend-connections on a social network and each user can connect only with the users they have indicated as friends. IPOP

[15] `https://openvpn.net`
[16] `http://ipop-project.org`

provides each user with their own view of the network with mappings between addresses and friends, and performs address translation in cases where addresses collide with those used by other friends. In GroupVPN, a single flat address-space is built among all users who belong to a group, and all members of the group can communicate with each other. GroupVPN also supports a bridging mode, where a virtual Ethernet broadcast domain is formed among the devices.

In table 3 we summarise our survey. By client-side mobility we mean that the side usually initiating connections is allowed to move. By server-side mobility we mean that the side usually responding to connections is allowed to move. We make no claims of whether a rendezvous-point or a home agent is allowed to move. With route optimizations we mean, that the protocol readily supports a mode in which messages do not need to travel through a central point. Komu *et al.* provide a more thorough survey on mobility management protocols in their recent paper [28].

5.2 How HIP would solve challenges related to Cloudlets

In section 3.8.2, we identified challenges posed on a roaming solution for Cloudlets. The update mechanism of HIP, as discussed in section 4.2.2 allows endpoints to change locators without a disconnection occurring at the application layer.

The update is a three-way procedure involving cryptographic signatures. An update can be run after the two hosts have been successfully authenticated using the base exchange. Successfully running the update procedure requires access to the cryptographic key used as HI. It can thus be said, that HIP prevents hijacking of the connection.

HIP encrypts all traffic by default using IPsec and its secure cryptographic algorithms. Thus, HIP preserves confidentiality and integrity.

Out of all the protocols included in our comparison in section 5.1, HIP was the only one that supported mobility without the need to set up some sort of static infrastructure. As long as the hosts learn about each others locators somehow, HIP supports direct end-to-end connections. This means both that optimal routes can easily be used, and that HIP is not dependent on external infrastructure.

The two remaining challenges, recovering from mobility events without undue delay, and the preserving of the underlying network's bandwidth and

delays, are related to performance. We found nothing in the design of the protocol itself that would prevent it from achieving these goals. On the other hand, performance is often a characteristic of a specific implementation. In chapter 6 we discuss our process of porting the HIP implementation HIPL to Android. In chapter 7 we perform experiments on our port to provide an answer to these two remaining challenges.

5.3 Deploying HIP with Cloudlets

In a Cloudlet infrastructure, the mobile client, the Cloudlet host, and the Cloudlet VM are all individual hosts with separate identities. In this section we discuss the different strategies to deploy HIP.

5.3.1 HIP between a mobile client and a Cloudlet host

Between a mobile client and a Cloudlet host, a HIP connection can serve various purposes. In section 2.1, we identified trust establishment as one of the steps in cyber foraging. The cryptographic identities can provide a means to trust establishment, since they provide a way to verify if the host we are communicating with is the one with which we intended to communicate, and as needed, block communications with other identities.

At least in Elijah, the client orchestrates the lifecycle of the Cloudlet VM, and does this by issuing commands to the Cloudlet host. The mobility support of HIP ensures that the commands issued by the mobile client are processed in the same session even when roaming between networks. Encryption and integrity checks also prevent outsiders from injecting commands through the same channel.

As HIP encrypts the communication, this also provides a means to prevent eavesdroppers on the network to learn about which overlay image, or which payload is being transferred over the network.

Should the Cloudlet VM be accessed through forwarded ports on the Cloudlet host, using a HIP tunnel between the client and the Cloudlet host would provide some security for such communication as well. However, if the host is compromized, HIP does not help.

5.3.2 HIP between a mobile client and a Cloudlet VM

HIP based connectivity provides the same kinds of advantages between the client and the Cloudlet VM as between the client and the Cloudlet host. The application benefits from the mobility features of HIP, should the mobile client move between networks, or the Cloudlet VM be migrated on another host.

When used between the client and the VM, HIP based connectivity assures that the communication is taking place with the right VM, assuming that opportunistic mode is not used, and that nobody is intercepting the communications.

HIP preserves confidentiality of the data plane from the network, but also from the operating system of the Cloudlet host. However, it may be possible for the hypervisor to intercept messages after decryption or before encryption, so HIP alone does not protect against all dangers.

5.3.3 HIP between two Cloudlet hosts

HIP enabled Cloudlet hosts can use HIP to securely address each other using HITs or LSIs. This way they can ensure that the source and destination hosts for VM migration are the correct ones. The encryption and integrity protection provided by IPsec ensure that the VM is not altered during migration and that potential eavesdroppers cannot learn about the contents, or likely even about the owner of the VM. For multihomed hosts, HIP provides fault tolerance, since it can switch to using alternative locators if the initially used connections fail.

5.3.4 HIP between a Cloudlet host and a cloud backend

HIP can also secure the transmission of a Cloudlet VM overlay image, when the client has supplied an URL to a HIP enabled hosting site, instead of supplying the image itself. HIP provides the same benefits to partial VM image transfers as it does to transferring complete images during migration.

5.4 Challenges in deployment

The main challenges of deploying HIP are the availability of HIP implementations, and the distribution of host identities.

5.4.1 Availability of implementations

We discussed the available HIP implementations in section 4.7. To the best of our knowledge, the port we made of HIPL is the only available implementation of HIP for Android today. We found no implementations for iOS or Windows Phone. As we found in our survey in section 4.7, the support for recent versions of desktop operating systems is also relatively weak. In order for HIP to truly benefit Cloudlets, a modern implementation of HIP is needed for all major operating systems.

5.4.2 Keys to Cloudlet VMs

The structure of Cloudlet VM images, as discussed in section 3.4, causes a challenge for distribution of the Host Identities. If the keys are included in a base image, all overlays that do not overwrite them will share the same identity.

If the HIs are included in the overlay, all copies of the same overlay will use the same identity. To use different HIs with different copies, we would have to construct a copy of the overlay for each identity separately. This is analogous to tagging and compiling purchased software for each client separately. It would also diminish a lot of the benefit from caching overlay images in the Cloudlet host.

Constructing both images without HIs, the VM's can generate new identities as they are started. The VM's would have unique identities, but we would not readily know what they are. The VM can try to contact a specific identity once it has generated its own, but then again we would need a separate copy of the overlay per identity to contact.

Another way is to have the VM construct its identity upon start, and have the Cloudlet host report the IP and a port for the VM to the client. The client would then contact the IP and port in opportunistic mode and then it would likely be communicating with the correct VM. This would be better if the Cloudlet host had a method of reporting the VM's generated identity to the client.

Another way to address this would be to modify the current Cloudlet host software implementation to accept an ordered series of additional patches to the image. The synthesis would need to be changed so that after applying the overlay to the base image, but before launching the VM, it would apply

the patches in order to the image. This way small changes, such as the HIs, could be delivered separately from the images.

Probably the most suitable way that we found would be for the client to generate the identity it wants the VM to use, and supply the HIs to the Cloudlet host. The host would then inject the HI to the client, e.g., by attaching an emulated storage device.

6 Porting HIP for Linux to Android

HIP for Linux (HIPL) is an open source implementation of the Host Identity Protocol (HIP), developed as part of the InfraHIP[17] project and targeted at the Linux operating system. Since Android is based on the Linux kernel, we believed it was relatively straightforward to port HIPL to Android. In this chapter, we describe our contribution to HIPL. The source code, along with our changes, can be found in the project's code repository[18].

6.1 Preliminary setting

In the beginning of the work, HIPL was working well on Linux, at least when compiled against the GNU compiler toolchain. HIPL has also been ported to an earlier version of Android, and an emulator image was available to prove this. The code, however, no longer compiled against the latest Android version, which was 4.3 (Jellybean) at the beginning of this work.

HIPL used the *ipqueue* packet queue for passing IP packets between kernel space and user space. *ipqueue* had been deprecated from the Linux kernel in favor of the Netfilter framework, and it was finally removed in kernel version 3.5. We changed HIPL to use *nfnetlink_queue* instead of the deprecated *ipqueue*, to ensure that HIPL would work in Android, but also to ensure it would continue working on desktop Linux distributions in the future.

6.2 Changes

To make HIPL work on the latest Android version, we made other changes as well. Android uses a Berkeley Software Distribution (BSD)-based C standard library called Bionic, that is different from the GNU C standard library (GLibC) used in most Linux distributions. HIPL relied on some features that were available in GLibC, but not in Bionic. Usually, however, we found an alternative function or structure that was available in both libraries.

[17] Infrastructure for HIP project: `http://infrahip.net`
[18] HIPL on Launchpad: `https://launchpad.net/hipl`

6.2.1 Re-definitions

Most of our changes involved defining or redefining macros, and could be isolated in the android/android.h file. This file is included in all files that use the definitions when compiling HIPL for Android.

```
1  #include "config.h"
2
3  #ifdef CONFIG_HIP_ANDROID
4  #ifndef HIPL_ANDROID_ANDROID_H
5  #define HIPL_ANDROID_ANDROID_H
6
7  #include <stdint.h>
8
9  /* Logging */
10 #define ALOGE printf
11
12 /* System properties */
13 #define PROPERTY_KEY_MAX 32
14
15 /* Networking */
16 #define ICMP6_FILTER 1
17 #define HOST_NAME_MAX 64
18 typedef uint16_t in_port_t;
19
20 #endif /* HIPL_ANDROID_ANDROID_H */
21 #endif /* CONFIG_HIP_ANDROID */
```

Listing 1: Missing definitions for constants

6.2.2 NetLink

HIPL uses the NetLink Transform (XFRM) subsystem to manage the IPsec SAs, and the relevant header file was not included in Bionic, so we borrowed it from the Linux kernel headers. The borrowed header is currently included with our source code and can be found under *android/linux/xfrm.h*.

HIPL also uses the *getifaddrs()* function, which is not included in Bionic. We borrowed an implementation of the function from Ken MacKay[19], who had just recently made it available to the public. Because of coding standards set for the HIPL code base, we made slight modifications to the source and shared back out changes with McKay. We modified the function to check

[19] `https://github.com/kmackay/android-ifaddrs`

for memory allocation errors, to inform the calling function about the errors, and changed the coding style to follow that of the rest of the HIPL source code.

6.2.3 ICMP6 filters

The ICMP6 protocol filter macros in Bionic were targeted for a BSD kernel, even though Android runs on the Linux kernel. We redefined the macros to work with the Linux kernel.

```
1  #undef ICMP6_FILTER
2  #undef ICMP6_FILTER_SETBLOCK
3  #undef ICMP6_FILTER_SETBLOCKALL
4  #undef ICMP6_FILTER_SETPASS
5  #undef ICMP6_FILTER_SETPASSALL
6  #undef ICMP6_FILTER_WILLBLOCK
7  #undef ICMP6_FILTER_WILLPASS
8
9  #define ICMP6_FILTER 1
10 #define ICMP6_FILTER_SETBLOCK(type, filterp)      \
11         ((((filterp)->icmp6_filt[(type) >> 5])    \
12         |=  (1 << ((type) & 31))))
13
14 #define ICMP6_FILTER_SETBLOCKALL(filterp)         \
15         memset (filterp, 0xFF,                    \
16                 sizeof(struct icmp6_filter));
17
18 #define ICMP6_FILTER_SETPASS(type, filterp)       \
19         ((((filterp)->icmp6_filt[(type) >> 5])    \
20         &= ~(1 << ((type) & 31))))
21
22 #define ICMP6_FILTER_SETPASSALL(filterp)          \
23         memset (filterp, 0x00,                    \
24                 sizeof(struct icmp6_filter));
25
26 #define ICMP6_FILTER_WILLBLOCK(type, filterp)     \
27         ((((filterp)->icmp6_filt[(type) >> 5])    \
28         & (1 << ((type) & 31))) != 0)
29
30 #define ICMP6_FILTER_WILLPASS(type, filterp)      \
31         ((((filterp)->icmp6_filt[(type) >> 5])    \
32         & (1 << ((type) & 31))) == 0)
```

Listing 2: ICMP6 Filter redefinitions

The heartbeat extension in HIPL uses these filters to catch replies to ICMP6 echo requests. The wrong definitions for the filters caused it to miss the replies, to determine that the connection must have failed and finally to tear down the SA related to the connection. This manifested in connections being systematically torn down approximately one minute after they were established, regardless of whether they were active or not. The definitions for the filter macros seem to be corrected since then for newer versions of Bionic[20].

6.2.4 Unavailable functions

HIPL originally used the *lockf* function to obtain locks on files. *lockf* was not available in Bionic, so we changed HIPL to use the *flock* function instead. There is a semantic difference between these functions that makes *lockf* preferable in most cases, but since we did not anticipate a need to store the files on a network file system, we find that the semantics of *flock* will suffice.

The functions for byte order conversions, *htons, htonl, ntohs* and *ntohl* are defined in GNU C as functions, but in Bionic as macros only. The HIPL source code originally contained a part where these functions were addressed in a selector function using function pointers. During compilation, the C preprocessor removes calls to macros and replaces them with the implementations defined in the macros. The macros will not have an address in the same way that functions do, and hence cannot be referenced by a pointer. We rewrote this part of HIPL into a form that no longer uses function pointers.

Bionic does not have implementations for certain functions that HIPL uses for lowering its privileges (or *capabilities*) when it no longer needs them. To address this, we changed the compilation configuration scripts to only compile the capability code if not compiling for Android.

6.2.5 Position Independent Executables

As of version 4.0, Android uses Address Space Layout Randomization (ASLR)[21], a mechanism for randomly choosing locations for objects in memory. Its purpose is to defend against application exploits that rely on knowing where

[20] `https://android.googlesource.com/platform/bionic/+/bfc6a59%5E!/`
[21] PaX ASLR: `http://pax.grsecurity.net/docs/aslr.txt`

something is located in memory. In version 4.1, support for Position Independent Executable (PIE), an ASLR technique in which executables consist only of position-independent code, was added to Android[22], and since version 5.0 Android has only accepted PIE executables[23].

The Android NDK supports compiling code as PIE through an added compiler flag *-pie*. Since PIE is the norm for new Android versions, we changed the HIPL compilation configuration scripts to use the flag by default when compiling for Android, and to accept a *–disable-android-pie* flag in case we are compiling for an older version of Android.

6.2.6 OpenSSL deprecated functions

HIPL was using the *RSA_generate_key* function to generate RSA keys to be used as Host Identities. The OpenSSL developers declared the function deprecated in OpenSSL 1.0.2[24]. We encountered that OpenSSL was compiled in Android with the *no-deprecated* flag enabled, resulting in a library where this function was not available. Since this issue affects other platforms as well, we changed HIPL to use the newer *RSA_generate_key_ex* function in all builds.

6.2.7 HIP Firewall

With the above changes, we were successful at compiling *hipd*, the base HIP protocol daemon. To compile *hipfw*, the daemon needed for firewall functionality and LSI support, some further changes were necessary. We made no changes to *hipfw* itself, but instead patched a number of changes to the libraries it was compiled against.

Three of our patches targeted the Android NDK. We patched *netinet/ip.h* to include definitions for addressing the Type of Service (TOS) field in IP packets. This definition has previously been removed from Bionic. We patched *netinet/ip_icmp.h* to include a definition for *struct icmphdr*, the format of the ICMP header. Finally we patched *linux/byteorder/swab.h* to include definitions for the 16, 32, and 64 bit versions of the *fswab* function, that is used in byte order conversions.

[22] `http://source.android.com/security/enhancements/enhancements41.html`
[23] `http://source.android.com/security/enhancements/enhancements50.html`
[24] `https://www.openssl.org/docs/man1.0.2/crypto/RSA_generate_key.html`

The Netfilter framework is used to transfer packets between user-space and kernel-space. Bionic defines a type called *tcp_word_hdr* which is also defined by *libnetfilter queue*. We addressed the collision of the types by removing the definition from *libnetfilter_queue* before compilation.

After these changes, we were also able to compile *hipfw*, and communicate with other hosts over LSIs. Porting of the third component, the HIPL DNS Proxy, used for intercepting HIs from DNS replies, was left for future work. The main reason for this was that the HIP DNS Proxy is implemented in Python, and porting it would have required a very different approach.

6.3 Compiling and running

To make compiling HIPL for Android easier, and especially to spare others from repeating our quest to find the right toolchain, we wrote a bash script that downloads and prepares a working build environment. The script can be found in the HIPL source code repository under the *tools* folder by the name *prepare_android_toolchain.sh*.

Currently, HIPL relies on a number of kernel modules and needs superuser access to the device. To use our port of HIPL, we needed to compile our own kernel. We included instructions on how to compile Android operating system, along with instructions for configuring and compiling a suitable kernel, in Appendix A. Instructions for compiling HIPL for Android from source code are included in Appendix B.

7 Experiments with HIP

In addition to functionality, performance is also important for deciding whether HIP makes for a suitable mobility solution for Cloudlets. High bandwidth and low round-trip times are some of the core features in Cloudlets. To support this, a good mobility solution would not sacrifice either, and would help the system to recover from mobility events in a reliable and swift manner. We designed a set of experiments to measure the performance of the HIP for Linux implementation in scenarios that we believe to be typical when using Cloudlets.

7.1 Experiment setup

Our experiment setup consisted of three servers, and two mobile devices. Two of the servers, *charlie* and *delta*, are HP ProLiant rack-mounted servers with a 1Gbps link to each other and to the Internet. The remote server *farnsworth* is a custom built PC connected to a 1Gbps switch, a 802.11n WiFi base-station, and a 100Mbps/10Mbps Internet connection. All the servers are running Ubuntu 12.04 and have the necessary software for running Cloudlets installed. Further specifications can be found in Table 4.

Computers				
Host	Model	CPU	RAM	Network
Charlie	HP ProLiant BL280c 6G	Intel Xeon E5640 4 cores, 2.66 GHz	8GB	1Gbps
Delta	HP ProLiant BL280c 6G	Intel Xeon E5640 4 cores, 2.66 GHz	8GB	1Gbps
Farnsworth	Custom	Intel Core i7-3770 4 cores, 3.40 GHz	32GB	1Gbps (100/10Mbps)
Mobile devices				
Host	Model	CPU	RAM	Network
Nexus	LG Nexus 5 (hammerhead)	Krait 400 4 cores, 2.26 GHz	2GB	LTE / 802.11n
Laptop	Acer Aspire One A110	Intel Atom N270 1 core, 1.60 GHz	1.5GB	802.11g

Table 4: Specifications of equipment

For the mobile device, we used a Google Nexus 5 smartphone developed by LG, running CyanogenMod with a customized kernel. Detailed instructions for the customization can be found in the appendices. For the wired tests, we connected an Apple USB Ethernet adapter (100Mbps) over a USB-OTG adapter cable.

7.2 Methodology

Our experiments focused on throughput and recovery from network migrations. For the throughput measurements, we measured the throughput reported by iperf or iperf3 over HIP connections with various configurations, and compared them with iperf-measurement results over plain IPv4 connections.

For the migration tests, we exchanged ICMP6 echo -messages (*ping6*) over the HIP connection while migrating the device at one end of the connection between hosts or access points. We measured both the interval between receiving replies, and the RTT on those replies.

For systematic results and to minimize human errors, we used bash-scripts to drive the experiments. We commanded the devices through virsh, SSH, D-Bus or ADB when needed.

7.3 Impact on throughput

Network traffic over a HIP connection is encrypted by the sender and decrypted by the receiver. This requires some additional computation compared to non-encrypted connections. Also, since the HIPL daemon performs its work in user space, this incurs some penalty in the form of copying packets back and forth between the packet queues in kernel space, and the memory of *hipd* and *hipfw* processes located in the user space. As with other tunneling solutions, HIPL also takes the penalty of a decreased MTU due to the use of additional headers.

We expect a slight penalty for throughput. Our expectation is supported by similar results by Osmani *et al.* who performed a throughput and latency comparison for a collection of tunnelling solutions, including HIP [48]. To measure the extent of this penalty, we performed a series of benchmarks.

7.3.1 Throughput between two servers

We performed benchmarks to examine the effect of employing HIPL on traffic throughput between the two identical servers *charlie* and *delta*. The throughput was measured using *iperf* in TCP mode. We used iperf in its default configuration, where the client uploads data and the server discards it.

We tested the throughput over various ways in which HIPL can be configured and used plain IPv4 as the benchmark. The average throughput between *charlie* and *delta* with plain IPv4 was 919 MBit/s, which we find satisfactory for a one-gigabit link. Running just hipd at both servers and performing the throughput test over the HITs reduced the average throughput to 383 MBit/s. Starting hipfw in the sending end reduced the average throughput even further down to just 93 MBit/s. Starting hipfw at the receiving end also reduced the throughput even further down to around 58 Mbit/s and having LSI's involved at either or both ends finally dropped the performance down to an average of 29 Mbit/s[25]. On the servers, hipfw was always started with the default flags, i.e. *-blkpF*.

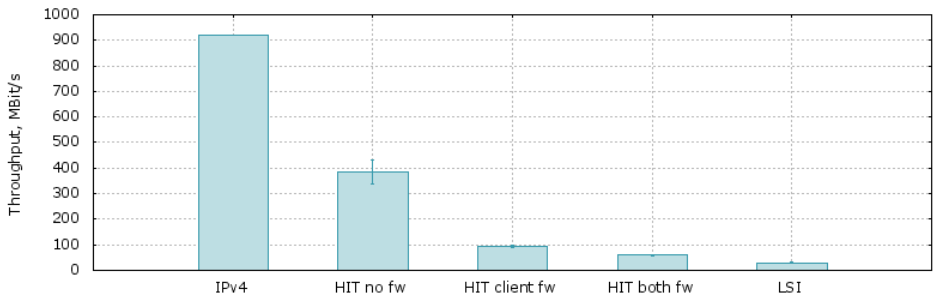

Figure 9: Throughput from iperf tests over various configurations

The throughput with most configurations was relatively stable with small standard deviations, but the throughput with HIP without hipfw seemed to fluctuate. A slight increase occurs in the standard deviation in all HIP configurations. Figure 9 illustrates the throughput the different experiment configurations.

7.3.2 Throughput for mobile device

We benchmarked the Android port of HIPL by running *iperf3* in both directions between a Nexus 5 and the host Farnsworth in multiple configurations. For the WiFi measurements, the Nexus 5 was connected to the network over an IEEE 802.11n wireless network and, for the wired setup, over an Apple USB Ethernet connector specified for 100Mbps. Benchmarks for plain IPv4

[25]The poor throughput of the LSI implementation was further inspected during the final stages of thesis. It appears that it can at least be doubled: `https://bugs.launchpad.net/hipl/+bug/1515296`

showed that throughput for the wireless network was better than for the wired network, but also more irregular. Figures 10 and 11 illustrate the throughput for wired and wireless measurements respectively.

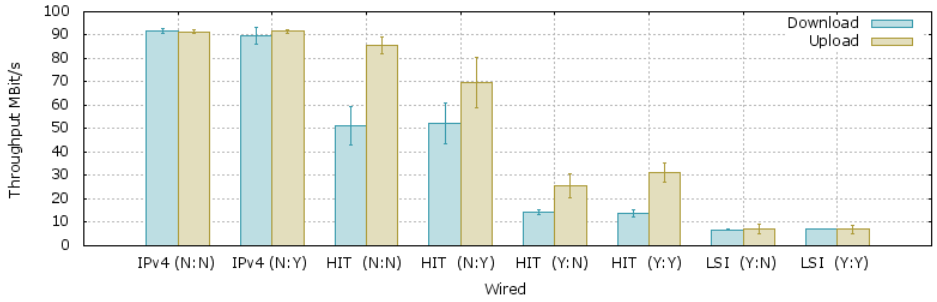

Figure 10: Download and upload throughput between Nexus 5 and Farnsworth respectively over Ethernet. The "Y:N" tells that hipfw was running on the Nexus, but not on Farnsworth.

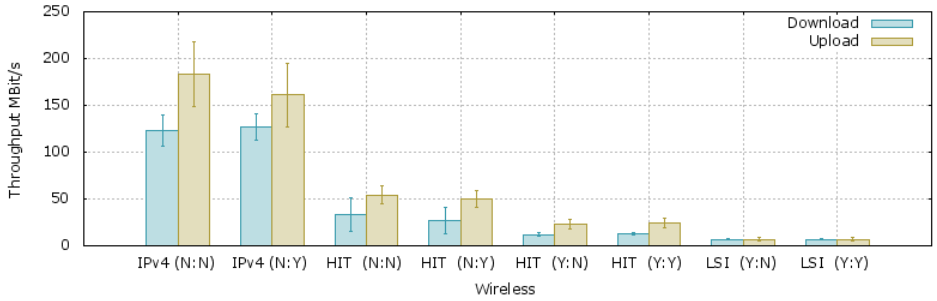

Figure 11: Download and upload throughput between Nexus 5 and Farnsworth respectively over WiFi.

As we expected, the throughput over HIP/IPsec was lower than over plain IPv4. On WiFi, the addition of HIPL reduced the throughput to roughly one quarter of the original, but this is still a tolerable rate for many applications. The addition of firewall functionality and especially LSI caused a grave degradation of performance. On wired connections, the performance seems steadier all around, and the degradation in performance before introducing firewall and LSI functionality was smaller. This may be affected by other wireless traffic competing for airtime, possible packet loss, or the fact that the link between our device and the WiFi access point is encrypted (WPA2) as well, while the wired link is not.

In these measurements, we started hipfw at the server with the default

flags -blkpF, and on the smartphone with the flags -blkF. The -p flag, for lowering privileges after iptables rules have been established, is not supported on Android. This should not affect performance.

7.4 Recovery from migration

In addition to throughput, a good mobility solution for Cloudlets also needs to be able to recover from mobility events, including migrations in a reasonable time. We believe that migration of the Cloudlet VM, and a network change by the mobile client are the most likely mobility events to occur, and hence we measured HIPL's performance in recovering from them. We have collected results from all migration measurements into Figure 16.

7.4.1 VM Migration

In the first experiments we migrated a KVM virtual machine between the hosts *charlie* and *delta*, and *ping6* was executed between the host *farnsworth* and the VM. We measured the time it takes for the VM to recognize that it has been moved and to recover the connection using the locator update procedure. Figure 12 shows the experiment setup. Our results can be seen in figures 13 and 16. We used a regular KVM virtual machine instead of a Cloudlet VM in Elijah because the modified QEMU used in Elijah did not support migrations at the time, and regular KVM VMs were our closest alternative.

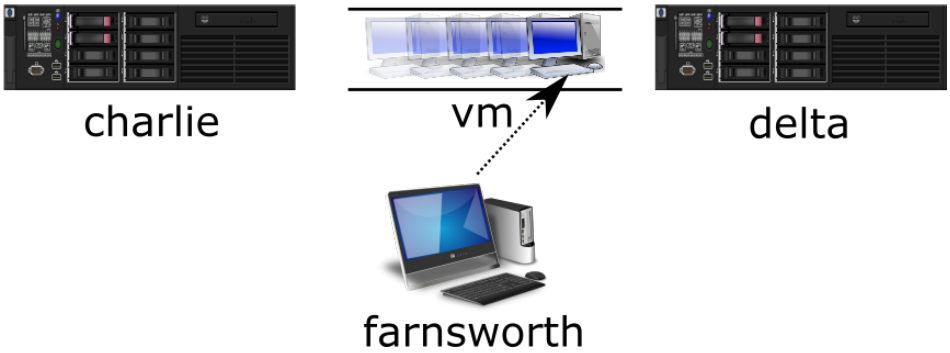

Figure 12: Recovery from VM migration experiment setup.

KVM / libvirt does not update the guest VM with changes in network topology. Hence, the time it takes to recover from a migration is largely based on timeouts that trigger the update or reconnection process. With the

current default configuration of HIPL, the average time it took to recover the connection was 27.89 seconds. During the live migration, the traffic included sending the VM image between the two servers, and this caused some variation in latency. Irregular lattices can also affect the usability of the Cloudlet VM, depending on the application. It is possible to mitigate this variation by transferring the VM over separate network, which implies more cost, or by setting a bandwidth limit for the migration, which would imply longer migration times.

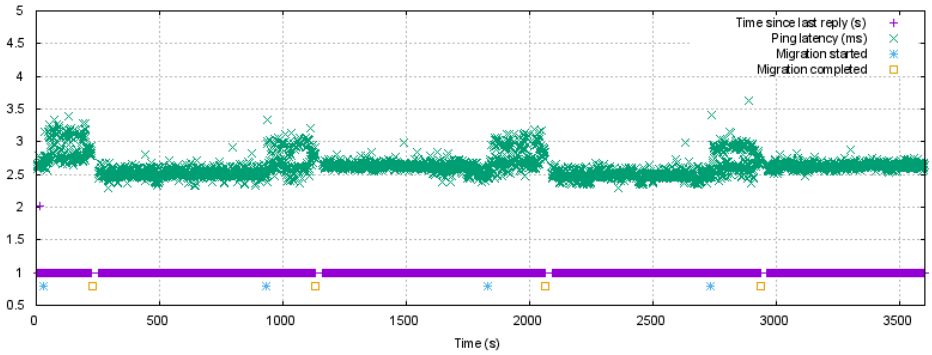

Figure 13: Recovery latency and RTTs during VM migrations.

7.4.2 Client Roaming - Laptop

In our second roaming experiment, we used a laptop computer to roam between two wireless networks while running *ping6* over a HIP tunnel between the laptop and the host *charlie* to see how long it takes to recover the connection. The experiment setup is depicted in Figure 14. We experimented with roaming between a WiFi network and a 3G hotspot, and with roaming between two WiFi networks. In the WiFi - 3G case, one of the WiFi networks was connected to a 100Mbps/10Mbps wired Internet connection, while the other was a mobile 3G connection. In the WiFi - WiFi case, we alternated between two visitor networks operated by University of Helsinki. The results are in Figures 15 and 16.

In this scenario, hipd was running on the same logical host as the bottommost network stack, i.e. there was no VMM that would hide the address change, like in the previous experiment. As the operating system could learn about the address change right away, HIPL was able to start the update procedure immediately after the locator change occurred. As Figure 16

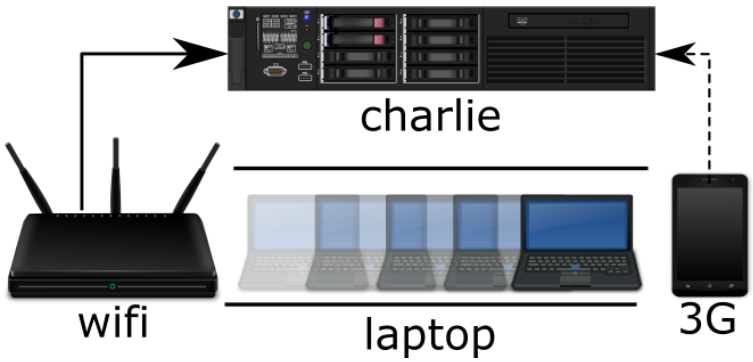

Figure 14: Experiment setup for client roaming between networks.

indicates, recovery was much quicker when the operating system could inform hipd when to start the update process. For Cloudlet use, a modification to some part of the Cloudlet software that would cause hipd to initiate the update process immediately after a migration has completed would greatly speed up connection recovery in the migrating VM scenario.

Roaming from 3G to WiFi was slightly quicker than the other way, with an average of about one second shorter time before recovery. The recovery time while roaming between the two WiFi networks was close to that of roaming to the 3G network, with relatively high amount of variation. We explain this by the networks not being exclusive to our experiment. Overall, in all scenarios the connection was recovered in roughly eight seconds on average.

7.4.3 Client Roaming - Smartphone

The experiments with the laptop provide a valuable reference point, but since our work focused on porting HIP for Linux to Android, we repeated the experiment on a Nexus 5 Android smartphone. We opened a HIP connection between the Nexus 5 and the host Farnsworth, and executed ping over the connection while periodically alternating between WiFi and LTE data connections.

After issuing the command to switch networks, the required time to recover the connection was typically between 5-12 seconds, averaging 8.19 seconds. The connection recovery performance is on par with the performance achieved on the laptop in the previous experiment. Figure 17 illustrates the latency related to hopping.

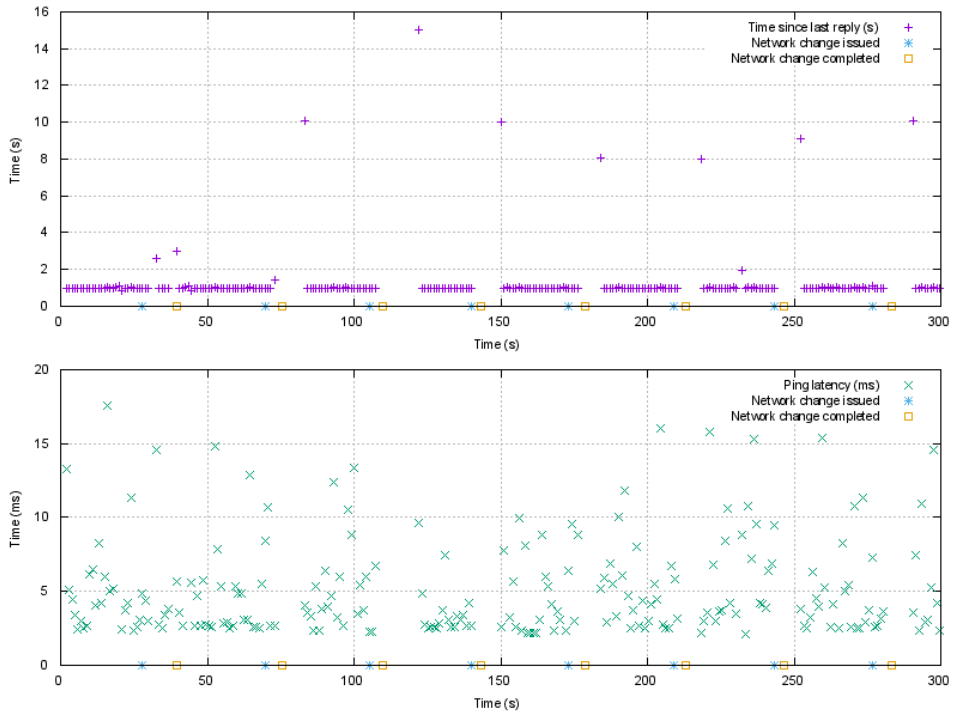

Figure 15: Latency when roaming a laptop between two WiFi networks. Above are the intervals between received messages, and below are the RTTs computed form the ping replies. The gaps in the intervals depict the recovery latencies.

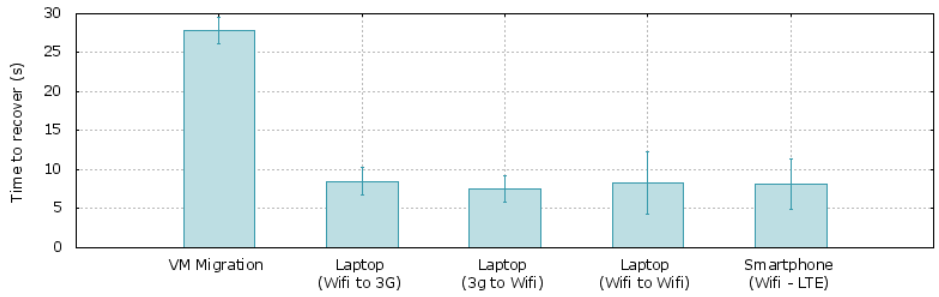

Figure 16: Recovery latency with different migrations.

With our experiment setup, while roaming from WiFi to LTE, the WiFi connection became unavailable immediately, but when roaming from LTE to WiFi, the LTE connection remained active until a WiFi association was established. We have addressed this in our analysis by ignoring the ping replies received between commanding the device to connect to WiFi and the

short gap in receiving replies.

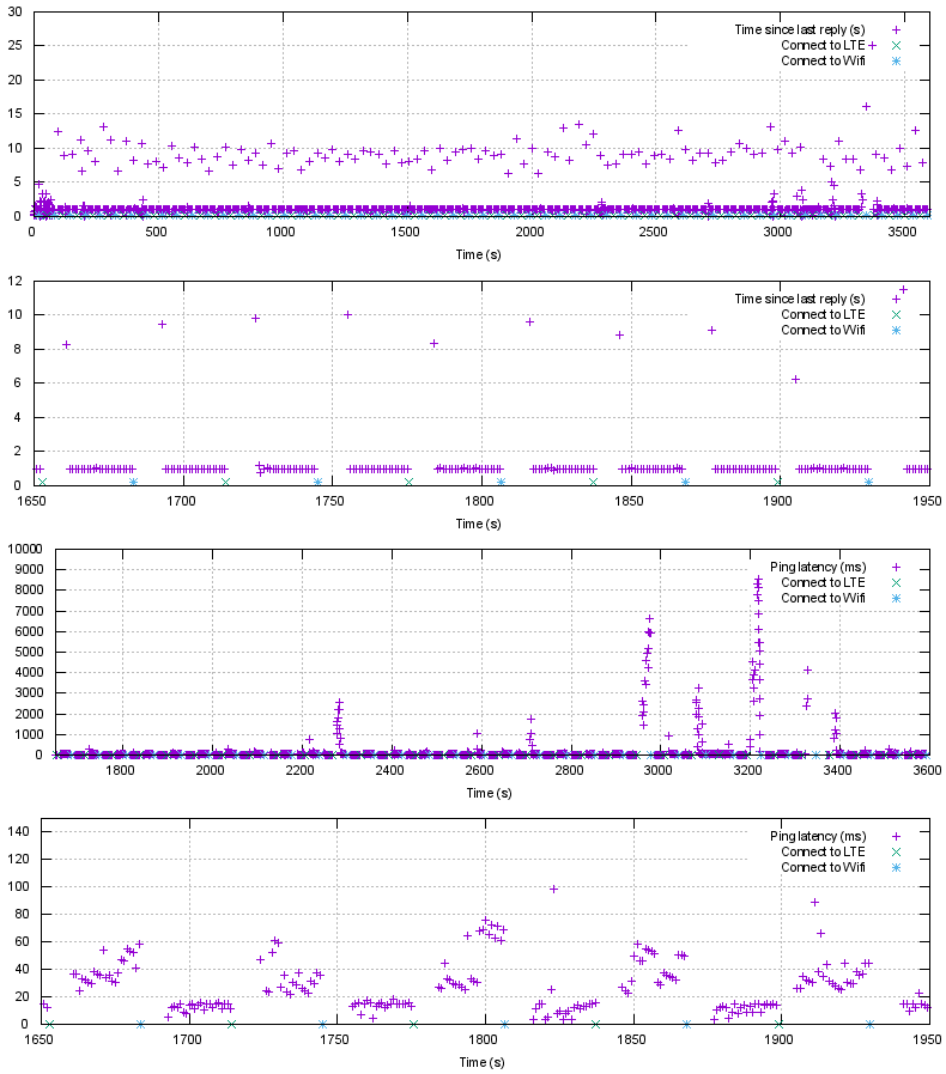

Figure 17: Recovery latency when migrating between WiFi and LTE

7.5 Migration vs reinstantiation

An interesting question is whether to migrate an existing Cloudlet VM or to instantiate a new one. The time required for Cloudlet synthesis to complete is largely dependent on the size of the overlay. The downtime at the end of a live migration while the SA is being updated is not. We assume that a production quality Cloudlet hosting platform would tell the Cloudlet VM of

a network status change, and hence our average downtime would be roughly eight seconds, following our results in section 7.4.

Ha *et al.* implemented a number of optimizations to Cloudlet synthesis, compare their performance to a baseline synthesis performance [18]. They evaluated five overlay images, of sizes between 0.5MB and 97.5MB. The smallest image is so small, that even the baseline synthesis time stays at 7.3 seconds. With the other overlays, the synthesis time varied between 37.0 and 140.2 seconds. With their full range of optimizations, Ha *et al.* were able to reduce the synthesis time of all but the largest overlay to between 5 and 10 seconds. The synthesis time of the largest overlay was reduced to roughly 48 seconds. It should be noted that the results for the fully optimized synthesis are the first-response times, i.e., only the most essential parts have been transfered and the VM has been started early while the synthesis process still continues in the background. It should also be noted that a new Cloudlet VM starts from a clean state, i.e., any state from the Cloudlet VM on the previous host is lost, if it's not separately stored somewhere. Live migration of a VM preserves the state of the VM, so the application doesn't need to start over.

The time it takes to synthesize a new Cloudlet VM can, in many cases, through optimizations, be reduced to roughly equal the time that the Cloudlet VM would be unavailable at the end of migration, thus rendering the downtime comparison mute to a large extent. Thus the focus shifts to other questions. How important is the preservation of state in the VM? Should the VM be reinstantiated to make the app more responsive earlier? How seamless can the instantiation of a new Cloudlet VM be made? Can the traffic caused by migrating the full VM be justified? The answers to these questions, we argue, largely depend on the specific application.

8 Related work

The related work in this field roughly divides into mobile offloading, execution environment, and mobility management related work. We have already discussed offloading frameworks in section 2. We also made a small survey into available protocols that would support mobility in section 5.1. Some execution environments are perhaps not specifically targeted at mobile computation offloading, but we found them interesting nonetheless.

Docker[26] is a system for creating and running Linux containers. Like Cloudlet images, Docker containers also consist of a base image, and a collection of patches similar to the overlay image. A Docker image consists of a collection of UnionFS file system images and metadata that describes what else is needed to use the image. A Docker container is not a full VM, but shares the hosting operating system's kernel, and is thus very resource friendly when compared to hypervisor-based virtualization. The sharing of kernel has meant that Docker is Linux-specific, and the OS X and Windows clients for Docker are actually running a VirtualBox[27] VM running Linux. Microsoft and Docker have recently announced partnership, and Windows Server 2016 will be shipped with a Docker-like feature called Windows Server Containers, which will share a Docker compatible API [56].

Vagrant[28] (not to be confused with Vagrant [40]) is a system for automating the construction of uniform VMs. Vagrant reads instructions from a configuration file in order to download a VM image, and then using provisioning tools such as Chef or Puppet, equips the VM with the desired set of software and configuration. Vagrant supports configuring multiple VM's within the same configuration file, enabling easy configuration of complete virtual infrastructures for various needs.

The distribution of resources in proximity of mobile devices in Cloudlets pays resemblance to an edge computing solution called Liquid Applications [47]. In Liquid Applications, the base stations in a cellular mobile network are equipped with computing elements, that can be used to process workloads such as help with video streaming, or provide context aware services, such as personalized advertisements on nearby billboards.

[26]Docker: `http://docker.io`
[27]Oracle VM VirtualBox: `https://www.virtualbox.org/`
[28]Vagrant: `https://www.vagrantup.com/`

9 Future work

The Android port of HIPL still needs work to become compatible with the default configurations of Android devices available on the market. Currently it needs a rooted device with a customized kernel, despite everything HIP needs being included in the vanilla Linux kernel. The current HIPL implementation would not work on the kernel that was distributed with any of the Android devices that were used for testing, usually because one or more of the following features was missing: a) IPsec BEET mode, b) Dummy net driver, c) Null crypto algorithms.

The kernels didn't include support for IPsec BEET mode tunnels, but most of them seemed to support traditional IPsec tunnels. This dependency could be solved in a couple of ways. IPsec in user space would remove the need to alter the kernel, but would result in some performance degradation. Convincing device vendors or Google to enable IPsec BEET mode in the kernel by default, and this would ensure this feature for future devices, but not for old devices. Aftermarket firmware projects like CyanogenMod[29] could bring this to tech-savvy users of older devices as well. Finally, this dependency could be removed by transitioning to some other data-plane encapsulation but this would break compatibility with HIP standards[30]. The approach that we would recommend is a combination of the first two; using IPsec in the kernel if it's found available, and falling back to user space IPsec if it's not.

Switching to the TUN/TAP virtual network adapter would release HIPL from dependence on the dummy network driver. However, at least since Android 4.0 (Ice Cream Sandwich) the TUN/TAP driver has been available, because Android's VPNService API depends on it. The switching would also in a sense be a step toward better portability, due to the driver's availability on other platforms.

The null crypto algorithm effectively means disabling encryption. End-users are unlikely to want this, and HIPL developers mostly use them for experimentation. However support for this was required in RFC5201 [43]. Disabling null encryption by default and providing an option to enable it during the HIPL build process would largely solve this dependency. Re-

[29] `http://www.cyanogenmod.org/`

[30] RFC7401 supports adding different transport protocols, but currently BEET ESP is the default.

searchers would still compile their own kernels, but others wouldn't have to.

To be successful, HIPL needs to be encapsulated into an Android-native wrapper and packaged into an *.apk* installation package. Once the dependencies on the kernel features have been solved, the daemon still requires superuser access to the device. The new Android VPN Application Programming Interface (API)[31] can be used to configure a virtual network interface without superuser access, but using the API would require a wrapper. Also, instead of using the *hipd* by itself, it should probably be converted into a native Android background process and access the HIP-related functions through the Android Native Development Kit (NDK).

As with many other protocols where clients communicate directly with each other, the plain HIP and IPsec are problematic with certain configurations of NAT. For instance, many cellular providers are moving their consumer-grade data connections behind carrier-grade NAT devices. Research on NAT traversal solutions has been done, but comparative studies and a well supported implementation of a NAT traversal solution for mobile devices may lower the threshold to adopt HIP.

Currently, if the base image for the overlay is not installed to the Cloudlet when a mobile device requests VM synthesis, the request will not succeed. However, it can be made possible to download the base image from a remote cloud backend. For the time being, no serious infrastructure for distributing the images has been deployed, and the simplest way to install a base image to a Cloudlet is to copy it there manually, and issue the 'import-base' command that registers it to the Elijah software. CMU provides Ubuntu-based sample images that can be used as a basic starting point. Cloudlets could be extended to use the same kind of a distribution infrastructure for base images as the one used by Docker[32]. In Docker's infrastructure, standard base images are stored in a centrally managed CDN. Users who wish to make Docker images start by declaring which base they use, and if their computer doesn't have it already, it will download the most appropriate image from the CDN. An overlay image always knows the fingerprint of the base image it was derived from and it can be used to identify which base image to download and register from the CDN.

[31] `http://developer.android.com/reference/android/net/VpnService.html`
[32] `https://docker.io`

It has been shown by others that the use of Cloudlets or other surrogates can reduce the energy consumption of tasks on the mobile device [36, 35]. In this thesis, we have studied the suitability of the HIP protocol for Cloudlets in terms of functionality and performance. In performance, we have benchmarked throughput and latency, yet omitted any study on HIPs effects on energy consumption. The severe performance degradation when using LSI's suggests that there is very likely room for improvement on energy consumption as well. In addition to simple comparative energy benchmarks, this area offers challenges such as designing an energy- and network aware migration planning and scheduling facility for the VMs.

For Cloudlets, we focused on the state where the Cloudlet system was when we started. It has since been further developed into an extension to OpenStack, with some improvements over the original design. Even though OpenStack can provide many improvements on the management of Cloudlets, the fundamental principles stay the same. In this thesis, we focused on the suitability of Host Identity Protocol to support mobility, but investigating the new OpenStack Cloudlet support would be a natural next step for this work. In their recent work, Osmani *et al.* have already tested HIP with scientific payloads in an OpenStack environment [48].

10 Conclusion

We have identified the requirements for a roaming solution for Cloudlets. We have also shown that HIP, by its features, fulfills all the qualitative requirements. We have ported HIPL for the Android platform, and through benchmarks, we have shown that our port mostly satisfies the quantitative requirements as well.

We have compared our results with measurements in other works, and conclude that whether to migrate a Cloudlet VM or instantiate a new one cannot be decided solely based on the time that the VM would be unavailable. With highly optimized Cloudlet VM synthesis, the synthesis time roughly equals the connection recovery time.

The HIPL Android port performs relatively well as long as the HIPL firewall, and especially the LSI support, is not used. HIPL would benefit from certain architectural restructuring. Despite our misfortune with LSI performance, it appears that HIP would serve as a roaming solution with Cloudlets. The decreased throughput from using IPsec may be addressed by using cryptographic acceleration hardware.

We have also identified some of the requirements for a more comprehensive mobility management framework for Cloudlets, while delimiting their further study outside of the scope of this thesis. Additionally, we have surveyed a number of frameworks suitable for computation offloading, researched different protocols with features close to what we were seeking for the Cloudlets, and discussed some recent works that closely relate to our topics at hand. We have also found some directions for future development and paths for future research.

11 Acknowledgements

Although this thesis carries my name, it is not solely due to my own effort that it was made possible. Many people have been in my support, in one way or another, during the time this has been under work.

First I would like to thank Miika Komu, my instructor for the thesis, for his insights, his support, the long debugging sessions with HIPL and the Android NDK, spaghetti at Täffä, and most of all, patience with seeing this project through. Despite changing jobs, Miika saw this project through.

I would also like to thank Professor Sasu Tarkoma, my supervisor, for the topic, for his guidance, also his patience, and for allowing me to also participate in other research projects and instructing students, on the side of the thesis work. The experience I gained is invaluable. And I thank Professor Asokan, for persuading him to stretch his patience just a little further, to allow Whispair, another research work, to become reality.

I wish to also thank Diego Biurrun, for holding up strict discipline in following coding conventions in HIPL. Even though this sometimes annoyed me quite a bit, I believe, that in the end, I also learned quite a lot.

I thank Yu Xiao, for our early conversations about Cloudlets and introduction to the topic, and Vivek Balakrishnan and Teemu Kämäräinen, for supporting my work while performing their own measurements with different protocols and uses for Cloudlets.

I want to thank Miia Mäntymäki, for putting up with all my quirks and jabbering about all kinds of computer stuff during a large part of my life.

I thank Seppo Hätönen for sharing the workload in the EASI-CLOUDS project, and understanding that sometimes I was busy with the thesis. And Samu Varjonen, for small bits of "shadow guidance" and our many discussions about HIP.

I want to also thank Jenny Tyrväinen and her two year old daughter Aino, for pushing me to work and pull the final straps of this thesis together. *"Juhani, kirjoita sinä gradu."* -Aino.

Finally I would like to thank my parents and other family members, for supporting me throughout my studies, and often eagerly questioning me about my plans to graduate.

References

[1] Abolfazli, Saeid, Sanaei, Zohreh, Ahmed, Erfan, Gani, Abdullah, and Buyya, Rajkumar: *Cloud-based augmentation for mobile devices: motivation, taxonomies, and open challenges.* Communications Surveys & Tutorials, IEEE, 16(1):337–368, 2014.

[2] Barré, Sébastien, Paasch, Christoph, and Bonaventure, Olivier: *Multipath TCP: from theory to practice.* In *NETWORKING 2011*, pages 444–457. Springer, 2011.

[3] Belalem, Ghalem, Bouamama, Samah, and Sekhri, Larbi: *An effective economic management of resources in cloud computing.* Journal of computers, 6(3):404–411, 2011.

[4] Bettstetter, Christian, Vögel, Hans Jörg, and Eberspächer, Jörg: *GSM phase 2+ general packet radio service GPRS: Architecture, protocols, and air interface.* Communications Surveys, IEEE, 2(3):2–14, 1999.

[5] Calheiros, Rodrigo N, Ranjan, Rajiv, Beloglazov, Anton, De Rose, César AF, and Buyya, Rajkumar: *CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms.* Software: Practice and Experience, 41(1):23–50, 2011.

[6] Chen, Kuan Ta, Huang, Chun Ying, Huang, Polly, and Lei, Chin Laung: *An Empirical Evaluation of TCP Performance in Online Games.* In *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '06, New York, NY, USA, 2006. ACM, ISBN 1-59593-380-8. `http://doi.acm.org/10.1145/1178823.1178830`.

[7] Chun, Byung Gon, Ihm, Sunghwan, Maniatis, Petros, Naik, Mayur, and Patti, Ashwin: *Clonecloud: elastic execution between mobile device and cloud.* In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[8] Clark, Christopher, Fraser, Keir, Hand, Steven, Hansen, Jacob Gorm, Jul, Eric, Limpach, Christian, Pratt, Ian, and Warfield, Andrew: *Live Migration of Virtual Machines.* In *Proceedings of the 2Nd Conference*

on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. `http://dl.acm.org/citation.cfm?id=1251203.1251223`.

[9] Clinch, Sarah, Harkes, Jan, Friday, Adrian, Davies, Nigel, and Satyanarayanan, Mahadev: *How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users.* In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pages 122–127. IEEE, 2012.

[10] Cuervo, Eduardo, Balasubramanian, Aruna, Cho, Dae ki, Wolman, Alec, Saroiu, Stefan, Chandra, Ranveer, and Bahl, Paramvir: *MAUI: making smartphones last longer with code offload.* In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[11] Feamster, Nick, Rexford, Jennifer, and Zegura, Ellen: *The road to SDN: an intellectual history of programmable networks.* ACM SIGCOMM Computer Communication Review, 44(2):87–98, 2014.

[12] Figueiredo, Renato, Jung, Youna, Juste, Pierre St., and Jeong, Kyuho: *IP over P2P (IPOP).* White Paper, July 2014. `http://ipop-project.org/wp-content/uploads/2014/07/IPOP-WhitePaper-1407.pdf`.

[13] Ford, A., Raiciu, C., Handley, M., and Bonaventure, O.: *TCP Extensions for Multipath Operation with Multiple Addresses.* RFC 6824 (Experimental), January 2013. `http://www.ietf.org/rfc/rfc6824.txt`.

[14] Fowley, Frank, Pahl, Claus, and Zhang, Li: *A comparison framework and review of service brokerage solutions for cloud architectures.* In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 137–149. Springer, 2014.

[15] Goldberg, Robert P: *Survey of virtual machine research.* Computer, 7(6):34–45, 1974.

[16] Gurtov, Andrei, Korzun, Dmitry, Lukyanenko, Andrey, and Nikander, Pekka: *Hi3: An efficient and secure networking architecture for mobile hosts.* Computer Communications, 31(10):2457–2467, 2008.

[17] Ha, Kiryong, Lewis, Grace, Simanta, Soumya, and Satyanarayanan, Mahadev: *Cloud offload in hostile environments.* Technical report, DTIC Document, 2011.

[18] Ha, Kiryong, Pillai, Padmanabhan, Richter, Wolfgang, Abe, Yoshihisa, and Satyanarayanan, Mahadev: *Just-in-time provisioning for cyber foraging.* In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 153–166. ACM, 2013.

[19] Huitema, C.: *Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs).* RFC 4380 (Proposed Standard), February 2006. `http://www.ietf.org/rfc/rfc4380.txt`, Updated by RFCs 5991, 6081.

[20] Ibrahim, Shadi, Jin, Hai, Cheng, Bin, Cao, Haijun, Wu, Song, and Qi, Li: *CLOUDLET: Towards Mapreduce Implementation on Virtual Machines.* In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 65–66, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-587-1. `http://doi.acm.org/10.1145/1551609.1551624`.

[21] Jalaparti, Virajith, Caesar, Matthew, Lee, Seungjoon, Pang, Jeffery, and Merwe, Jacobus Van der: *SMOG: a cloud platform for seamless wide area migration of online games.* In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, page 9. IEEE Press, 2012.

[22] Jiang, S., Guo, D., and Carpenter, B.: *An Incremental Carrier-Grade NAT (CGN) for IPv6 Transition.* RFC 6264 (Informational), June 2011. `http://www.ietf.org/rfc/rfc6264.txt`.

[23] Jokela, P., Moskowitz, R., and Melen, J.: *Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP).* RFC 7402 (Proposed Standard), April 2015. `http://www.ietf.org/rfc/rfc7402.txt`.

[24] Kämäräinen, Teemu: *Design, Implementation and Evaluation of a Distributed Mobile Cloud Gaming System.* Master's thesis, Aalto University, 2014.

[25] Kemp, Roelof, Palmer, Nicholas, Kielmann, Thilo, and Bal, Henri: *Cuckoo: a computation offloading framework for smartphones.* In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[26] Kim, Kibae, Kang, Songhee, and Altmann, Jorn: *Cloud Goliath Versus a Federation of Cloud Davids: Survey of Economic Theories on Cloud Federation.* TEMEP Discussion Papers 2014117, Seoul National University; Technology Management, Economics, and Policy Program (TEMEP), July 2014. `https://ideas.repec.org/p/snv/dp2009/2014117.html`.

[27] Kompella, K. and Rekhter, Y.: *Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling.* RFC 4761 (Proposed Standard), January 2007. `http://www.ietf.org/rfc/rfc4761.txt`, Updated by RFC 5462.

[28] Komu, Miika, Sethi, Mohit, and Beijar, Nicklas: *A survey of identifier–locator split addressing architectures.* Computer Science Review, 2015.

[29] Komu, Miika, Tarkoma, Sasu, and Lukyanenko, Andrey: *Mitigation of unsolicited traffic across domains with host identities and puzzles.* In *Information Security Technology for Applications*, pages 33–48. Springer, 2012.

[30] Kosta, Sokol, Aucinas, Andrius, Hui, Pan, Mortier, Richard, and Zhang, Xinwen: *Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading.* In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[31] Koukoumidis, Emmanouil, Lymberopoulos, Dimitrios, Strauss, Karin, Liu, Jie, and Burger, Doug: *Pocket Cloudlets.* SIGARCH Comput. Archit. News, 39(1):171–184, March 2011, ISSN 0163-5964. `http://doi.acm.org/10.1145/1961295.1950387`.

[32] Kozuch, Michael and Satyanarayanan, Mahadev: *Internet suspend/resume.* In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 40–46. IEEE, 2002.

[33] Krawczyk, H., Bellare, M., and Canetti, R.: *HMAC: Keyed-Hashing for Message Authentication.* RFC 2104 (Informational), February 1997. `http://www.ietf.org/rfc/rfc2104.txt`, Updated by RFC 6151.

[34] Kristensen, Mads Darø: *Scavenger: Transparent development of efficient cyber foraging applications.* In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 217–226. IEEE, 2010.

[35] Kumar, Karthik, Liu, Jibang, Lu, Yung Hsiang, and Bhargava, Bharat: *A survey of computation offloading for mobile systems.* Mobile Networks and Applications, 18(1):129–140, 2013.

[36] Kumar, Karthik and Lu, Yung Hsiang: *Cloud computing for mobile users: Can offloading computation save energy?* Computer, (4):51–56, 2010.

[37] Kuptsov, Dmitriy: *Implementing CuteHIP: Feasibility Analysis of Java-based Network-layer Security Protocols.* Finland: Aalto University.

[38] Laganier, J. and Eggert, L.: *Host Identity Protocol (HIP) Rendezvous Extension.* RFC 5204 (Experimental), June 2015. `https://tools.ietf.org/html/draft-ietf-hip-rfc5204-bis-06`.

[39] Lin, Tao and Wang, Shuhui: *Cloudlet-screen computing: a multi-core-based, cloud-computing-oriented, traditional-computing-compatible parallel computing paradigm for the masses.* In *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on*, pages 1805–1808. IEEE, 2009.

[40] Liu, Pengcheng, Yang, Ziye, Song, Xiang, Zhou, Yixun, Chen, Haibo, and Zang, Binyu: *Heterogeneous live migration of virtual machines.* In *International Workshop on Virtualization Technology (IWVT08)*, 2008.

[41] Moskowitz, R., Heer, T., Jokela, P., and Henderson, T.: *Host Identity Protocol Version 2 (HIPv2).* RFC 7401 (Proposed Standard), April 2015. `http://www.ietf.org/rfc/rfc7401.txt`.

[42] Moskowitz, R. and Nikander, P.: *Host Identity Protocol (HIP) Architecture.* RFC 4423 (Informational), May 2006. `http://www.ietf.org/rfc/rfc4423.txt`.

[43] Moskowitz, R., Nikander, P., Jokela, P., and Henderson, T.: *Host Identity Protocol.* RFC 5201 (Experimental), April 2008. `http://`

www.ietf.org/rfc/rfc5201.txt, Obsoleted by RFC 7401, updated by RFC 6253.

[44] Nikander, P. and Laganier, J.: *Host Identity Protocol (HIP) Domain Name System (DNS) Extensions*. RFC 5205 (Experimental), April 2008. http://www.ietf.org/rfc/rfc5205.txt.

[45] Nikander, P., Laganier, J., and Dupont, F.: *An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers (ORCHID)*. RFC 4843 (Experimental), April 2007. http://www.ietf.org/rfc/rfc4843.txt, Obsoleted by RFC 7343.

[46] Nikander, Pekka, Gurtov, Andrei, and Henderson, Thomas R: *Host identity protocol (HIP): Connectivity, mobility, multi-homing, security, and privacy over IPv4 and IPv6 networks*. Communications Surveys & Tutorials, IEEE, 12(2):186–204, 2010.

[47] NSN: *Nokia Solutions and Networks Intelligent base stations*. White Paper, November 2013. http://nsn.com/sites/default/files/document/nsn_intelligent_base_stations_white_paper.pdf.

[48] Osmani, Lirim, Toor, Salman, Komu, Miika, Kortelainen, Matti J., Lindén, Tomas, White, John, Khan, Rasib, Eerola, Paula, and Tarkoma, Sasu: *Secure Cloud Connectivity for Scientific Applications*. IEEE Transactions on Services Computing, Special issue on Cloud Services Meet Big Data, 2015.

[49] Perkins, C.: *IP Mobility Support for IPv4, Revised*. RFC 5944 (Proposed Standard), November 2010. http://www.ietf.org/rfc/rfc5944.txt.

[50] Perkins, C., Johnson, D., and Arkko, J.: *Mobility Support in IPv6*. RFC 6275 (Proposed Standard), July 2011. http://www.ietf.org/rfc/rfc6275.txt.

[51] Perkins, Charles E. and Johnson, David B.: *Route Optimization for Mobile IP*. Cluster Computing, 1(2):161–176, April 1998, ISSN 1386-7857. http://dx.doi.org/10.1023/A:1019033431871.

[52] Porras, Jari, Riva, Oriana, and Kristensen, Mads Darø: *Dynamic resource management and cyber foraging.* In *Middleware for Network Eccentric and Mobile Applications*, pages 349–368. Springer, 2009.

[53] Porras, Jari, Riva, Oriana, and Kristensen, Mads Darø: *Dynamic Resource Management and Cyber Foraging.* In Garbinato, Benoît, Miranda, Hugo, and Rodrigues, Luís (editors): *Middleware for Network Eccentric and Mobile Applications*, pages 349–368. Springer, 2009.

[54] Postel, J.: *Internet Protocol.* RFC 791 (INTERNET STANDARD), September 1981. `http://www.ietf.org/rfc/rfc791.txt`, Updated by RFCs 1349, 2474, 6864.

[55] Postel, J.: *Transmission Control Protocol.* RFC 793 (INTERNET STANDARD), September 1981. `http://www.ietf.org/rfc/rfc793.txt`, Updated by RFCs 1122, 3168, 6093, 6528.

[56] Russinovich, Mark: *Containers: Docker, Windows and Trends.* `https://azure.microsoft.com/en-us/blog/containers-docker-windows-and-trends/`. Accessed 2015-10-29.

[57] Saarinen, Aki, Siekkinen, Matti, Xiao, Yu, Nurminen, Jukka K, Kemppainen, Matti, and Hui, Pan: *Smartdiet: offloading popular apps to save energy.* ACM SIGCOMM Computer Communication Review, 42(4):297–298, 2012.

[58] Saint-Andre, P.: *Extensible Messaging and Presence Protocol (XMPP): Core.* RFC 6120 (Proposed Standard), March 2011. `http://www.ietf.org/rfc/rfc6120.txt`, Updated by RFC 7590.

[59] Satyanarayanan, Mahadev: *Pervasive computing: Vision and challenges.* Personal Communications, IEEE, 8(4):10–17, 2001.

[60] Satyanarayanan, Mahadev, Bahl, Paramvir, Caceres, Ramón, and Davies, Nigel: *The case for VM-based Cloudlets in mobile computing.* Pervasive Computing, IEEE, 8(4):14–23, 2009.

[61] Satyanarayanan, Mahadev, Gilbert, Benjamin, Toups, Matt, Tolia, Niraj, O'Hallaron, David R, Surie, Ajay, Wolbach, Adam, Harkes, Jan,

Perrig, Adrian, Farber, David J, *et al.*: *Pervasive personal computing in an internet suspend/resume system.* Internet Computing, IEEE, 11(2):16–25, 2007.

[62] Satyanarayanan, Mahadev, Smaldone, Stephen, Gilbert, Benjamin, Harkes, Jan, and Iftode, Liviu: *Bringing the cloud down to earth: Transient pcs everywhere.* In *Mobile Computing, Applications, and Services*, pages 315–322. Springer, 2012.

[63] Simanta, Soumya, Ha, Kiryong, Lewis, Grace, Morris, Ed, and Satyanarayanan, Mahadev: *A Reference Architecture for Mobile Code Offload in Hostile Environments.* In *Mobile Computing, Applications, and Services*, pages 274–293. Springer, 2013.

[64] Stoica, Ion, Adkins, Daniel, Zhuang, Shelley, Shenker, Scott, and Surana, Sonesh: *Internet indirection infrastructure.* In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 73–86. ACM, 2002.

[65] Varjonen, Samu, Komu, Miika, and Gurtov, Andrei: *Secure and efficient IPv4/IPv6 handovers using host-based identifier-locator split.* In *Software, Telecommunications & Computer Networks, 2009. SoftCOM 2009. 17th International Conference on*, pages 111–115. IEEE, 2009.

[66] Winstein, Keith and Balakrishnan, Hari: *Mosh: An Interactive Remote Shell for Mobile Clients.* In *USENIX Annual Technical Conference*, pages 177–182, 2012.

[67] Wolbach, Adam, Harkes, Jan, Chellappa, Srinivas, and Satyanarayanan, Mahadev: *Transient customization of mobile computing infrastructure.* In *Proceedings of the First Workshop on Virtualization in Mobile Computing*, pages 37–41. ACM, 2008.

ADB	Android Debugging Bridge
API	Application Programming Interface
AR	Augmented Reality
AS	Autonomous System
ASLR	Address Space Layout Randomization
BEET	Bound End-to-End Tunnel
BSD	Berkeley Software Distribution
CDN	Content Delivery Network
CIA	Confidentiality, Integrity, Availability
CLR	Common Language Runtime
CMU	Carnegie Mellon University
COTS	commercial off-the-shelf
DNS	Domain Name System
DoS	Denial of Service
ESP	Encapsulating Security Payload
HA	Host Association
HI	Host Identity
HIP	Host Identity Protocol
HIPL	HIP for Linux
HIT	Host Identity Tag
HMAC	Keyed-Hash Message Authentication Code
IaaS	Infrastructure as a Service
I/O	input/output
IPv4	Internet Protocol version 4

IPv6	Internet Protocol version 6
IP	Internet Protocol
L3	Layer 3 - Network layer in the TCP/IP protocol stack
LAN	Local Area Network
LSI	Local Scope Identifier
LTE	Long Term Evolution
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NDK	Native Development Kit
ORCHID	Overlay Routable Cryptographic Hash Identifier
PIE	Position Independent Executable
RPC	Remote Procedure Call
RVS	Rendezvous
SA	Security Association
SDN	Software Defined Networking
SSH	Secure Shell
SPI	Security Parameter Index
TCP	Transmission Control Protocol
TLD	Top-Level Domain
TOS	Type of Service
URL	Universal Resource Locator
VES	Virtual Execution System
VM	Virtual Machine
VMM	Virtual Machine Monitor

VNC	Virtual Network Computing
VPLS	Virtual Private LAN Service
VPN	Virtual Private Networking
WLAN	Wireless Local Area Network
XMPP	Extensible Messaging and Presence Protocol

A The Android Open Source Project

A.1 Building your own AOSP

A.1.1 Repo

AOSP uses a tool called *repo* for source code management. We need to download and install this tool first.

```
$ mkdir ~/bin
$ PATH=~/bin:${PATH}
$ curl \
    http://commondatastorage.googleapis.com/git-repo-downloads/repo \
    > ~/bin/repo
$ chmod a+x ~/bin/repo
```

A.1.2 Download AOSP

This will download the Android Open Source Project source code. It is quite big and this step takes time. Consider heading to lunch after launching repo sync.

```
$ mkdir aosp
$ cd aosp
~/aosp$ repo init -u \
  https://android.googlesource.com/platform/manifest
~/aosp$ repo sync
```

A.1.3 Download and unpack driver binaries

Drivers for many devices are proprietary and are not distributed within AOSP; We need to download them, unpack them and accept their licences. AOSP build process will search for proprietary drivers under directory vendor in the source root. For the Nexus device-series the right drivers can be found at `https://developers.google.com/android/nexus/drivers`.

```
~/aosp$ mkdir temp
~/aosp$ cd temp

# Download the driver binaries
```

```
~/aosp/temp$ for driver in \
    broadcom-maguro-jwr66y-5fa7715b.tgz     \
    imgtec-maguro-jwr66y-b0a4a1ef.tgz       \
    invensense-maguro-jwr66y-e0d2e531.tgz   \
    nxp-maguro-jwr66y-d8ac2804.tgz          \
    samsung-maguro-jwr66y-fb8f93b6.tgz      \
    widevine-maguro-jwr66y-c49927ce.tgz;
    do wget https://dl.google.com/dl/android/aosp/${driver};
    done

# Unpack each package
~/aosp/temp$ for package in *.tgz;
    do tar xf ${package};
    done

# and run each script.
~/aosp/temp$ for script in *.sh;
    do sh ${script};
    done

~/aosp/temp$ mv vendor ../vendor
~/aosp/temp$ cd ..
~/aosp$ rm -rf temp
```

A.1.4 Configure

These scripts prepare your build environment. Launching lunch without arguments will print a menu from which you can choose your build target. The OUT environment variable that well use later comes from here.

```
~/aosp$ source build/envsetup.sh
~/aosp$ lunch  full_maguro
```

A.1.5 Build

This will build the AOSP and put the result in OUT. Adjust the number after -j (number of compile threads) to your liking, a good rule of thumb is the number of cores +1. After this step it is possible to flash the phone with

the default AOSP system by booting your phone to bootloader and running fastboot -w flashall, but now is a bit early if we want the custom kernel.

```
~/aosp$ make -j4
```

At the time of writing, there were a couple of Makefiles that would get upset if you had the environment variable NDK_ROOT set. If you encounter this, simply 'unset NDK_ROOT' and try again.

A.2 Custom kernel

For HIPD to work, we need certain features from the kernel. These features are readily available but disabled by default in most Android kernels. Therefore we need to compile our own kernel with these features enabled.

A.2.1 Download

We start by downloading the kernel sources. For the Nexus series you can check which version you need from `http://source.android.com/source/building-kernels.html`. After cloning you can check which brances are available for checkout with git branch -a.

```
~/aosp$ mkdir ../kernel
~/aosp$ cd ../kernel
~/kernel$ git clone \
    https://android.googlesource.com/kernel/omap.git
~/kernel$ cd omap
~/kernel/omap$ git checkout \
    remotes/origin/android-omap-tuna-3.0-jb-mr2
```

A.2.2 Configure

We then tell the build-environment about the target architecture, add the compilers to PATH, load the default kernel configuration for our device and bring up a menu where we can do further configuration.

```
~/kernel/omap$ export ARCH=arm
~/kernel/omap$ export SUBARCH=arm
~/kernel/omap$ export CROSS_COMPILE=arm-eabi-
```

```
~/kernel/omap$ export PATH=$(pwd)/../../aosp/prebuilts/linux-x86/arm/\
    arm-eabi-4.6/bin:$PATH
~/kernel/omap$ make tuna_defconfig
~/kernel/omap$ make menuconfig
```

In the configuration menu, enable the following:

```
- Enable loadable module support
- Networking support > Networking options > IP: IPsec BEET mode
- Device drivers > Network device support > Dummy net driver support
- Cryptographic API > Null algorithms
```

A.2.3 Build

Once the kernel has been configured, its time to build it.

```
~/kernel/omap$ make
```

If there are problems with smc #0, you can try adding .arch_extension sec to the offending files. (I used sed -e 1i.arch_extension sec -i filename, it worked, but Im not sure if its the right way.)

If there is a missing elf.h, you can copy it from under the aosp directory in external/elfutils/libelf/elf.h

A.2.4 Copy the kernel to AOSP

When the kernel has been compiled, we need a way to put it on the device. We replace the original AOSP kernel and put its modules in place. Make produces two symbolic links that we dont want present in the final image.

```
~/kernel/omap$ cp arch/arm/boot/zImage $OUT/kernel
~/kernel/omap$ make INSTALL_MOD_PATH=$OUT/system modules_install
~/kernel/omap$ rm $OUT/system/lib/modules/*/source
~/kernel/omap$ rm $OUT/system/lib/modules/*/build
```

A.2.5 Rebuild boot image

Now that our kernel is in place in the AOSP tree, we need to rebuild the boot image with the new kernel.

```
~/kernel/omap$ cd ../../aosp
~/aosp$ make bootimage
```

A.3 Installing the new AOSP on device

A.3.1 Flashing the OS

Once the new images are complete, its time to flash them on the device. New images can be flashed from the bootloader (fastboot mode). You can generally get there with adb, but there is a manual way too: for the Nexus devices `http://source.android.com/source/running.html`. If the device has a locked bootloader, it needs to be opened (usually fastboot oem unlock). (this might prevent some DRM from working!) Some of the steps might reboot the phone; the important part is that all the fastboot commands need to be entered with the device in bootloader.

```
~/aosp$ adb reboot bootloader
~/aosp$ fastboot oem unlock
~/aosp$ fastboot reboot-bootloader
~/aosp$ fastboot -w flashall
~/aosp$ fastboot reboot
```

The -w in fastboot -w flashall is important. It clears caches and previous user data that would confuse the new image and prevent it from booting all the way. If the phone doesnt boot, this is the first thing you should check.

A.3.2 Google apps (optional)

Now you have a clear Android open source operating system with our customised kernel. This means that proprietary apps and services like Google Play and contacts sync are not there. The easiest way to get them there is to install a custom recovery image, like the ClockworkMod recovery and flash it to the device using fastboot.

```
$ wget http://download2.clockworkmod.com/recoveries/
      recovery-clockwork-touch-6.0.4.3-maguro.img
$ adb reboot bootloader
$ fastboot flash recovery recovery-clockwork-touch-6.0.4.3-maguro.img
```

Once a recovery image is installed, you can boot into recovery mode either straight from the bootloader menu or from normally booted Android by adb reboot recovery.

B HIPL Android port

HIPL currently has partial experimental support for the Android platform. The parts that currently work are hipd, the HIP daemon; and hipconf, the configuration tool. hipd does require root privileges, and your running kernel must support the IPsec BEET mode, the dummy network driver and the null crypto algorithm. Often one or more of these are not compiled into your stock kernel, and it is likely that you need to compile and install your own kernel. On our development devices we went ahead and compiled the whole OS; this procedure is documented in another appendix of the work.

B.1 Compiling

Currently we only support compiling under Linux. We provide a script in *tools/prepare_android_toolchain.sh* that downloads and extracts the toolchain needed to compile hipd and hipconf and it has been confirmed to work at least on Ubuntu 12.04.

After downloading the HIPL source code toolchain is installed, the steps to compile for Android are almost similar to normal Linux builds. You start with autoreconf –install; then run the configure script:

```
$ ./configure
    --enable-android                                        \
    --disable-firewall                                      \
    --host=arm-linux                                        \
    --prefix=/usr                                           \
    --sysconfdir=/etc                                       \
    CC=${ANDROID_TOOLCHAIN}/bin/arm-linux-androideabi-gcc \
    CFLAGS="-std=c99 -mbionic -fPIC -fno-exceptions       \
            --sysroot=${ANDROID_SYSROOT}"                   \
    LDFLAGS="-Wl,-rpath-link=${ANDROID_SYSROOT}/usr/lib,-L\
    ${ANDROID_SYSROOT}/usr/lib"                             \
    LIBS="-lc -lm -lgcc -lcrypto"
$ make
```

B.2 Installing

After the build process completes, open a root privileged shell session on your phone and set up the environment:

```
$ adb root
$ adb shell
# mount -o remount,rw /
# mount -o remount,rw /system
# mkdir -p /var/lock
# mkdir    /etc/hip
# ln -s /system/lib/libcrypto.so /system/lib/libcrypto.so.1.0.0
```

Afterwards, you can return to your normal terminal and push hipd, hipconf and the configuration into the device:

```
$ adb push tools/hipconf   /system/xbin
$ adb push hipd/hipd       /system/xbin
$ adb push hipd/hipd.conf  /etc/hip
$ adb push hipd/relay.conf /etc/hip
```

B.3 Launching

It should now be possible to launch hipd.

```
$ adb shell
# hipd -ab
```

Since at the time of writing this only hipd and hipconf from the HIPL package has been successfully ported, LSIs and DNS based extensions are not available. Hosts can be configured either in the /etc/hosts file or introduced as added mappings in /etc/hip/hipd.conf.

On Android, hipd needs to be run with the '-a' parameter. Additionally it supports the same parameters as the normal Linux version does, i.e. '-k' kills an already running instance and '-b' starts hipd in the background.

B.4 Reboots

As the root file system on Android typically resides on a ramdisk, the /var/lock folder is removed every time the phone is restarted. For hipd to run, it needs to be recreated by running:

```
$ adb root
$ adb shell mount -o remount,rw /
$ abd shell mkdir -p /var/lock
```

To not need to do this on every reboot, HIPL can be patched to use a different file for locking on Android, or the initial ramdisk in Android can be changed to include either a writable /var/lock or a script that runs the commands on start.

/system and /etc typically reside in persistent storage, so the binaries and configuration files are safe.