VIRTUAL PLATFORMS: SYSTEM SUPPORT TO ENRICH THE FUNCTIONALITY OF END CLIENT DEVICES

A Thesis
Presented to
The Academic Faculty

by

Minsung Jang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2015

VIRTUAL PLATFORMS: SYSTEM SUPPORT TO ENRICH THE FUNCTIONALITY OF END CLIENT DEVICES

Approved by:

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Omesh Tickoo
Computer Vision Research
Intel Corporation

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Dr. Padmanabhan Pillai
Intel Research Pittsburgh
Intel Corporation

Date Approved: May 13, 2015

To my wife, Yoonhwa Hwang.

ACKNOWLEDGEMENTS

It has been an honor, privilege and pleasure to pursue a Ph.D. for several years at the Georgia Institute of Technology. It only seems like yesterday when I first started my degree, but almost seven years have passed and in a couple of months, I will be off to the next destination of my life.

I have a lot of people that have helped me out, both personally and professionally, during my years at Georgia Tech. First and foremost, my advisor, Professor Karsten Schwan, has been supporting me over the years. With his invaluable mentorship on the research, I have been able to achieve research outcomes. I also owe Dr. Ada Gavrilovska thanks.

I would like to thank the remaining members of my dissertation committee, Professor Umakishore Ramachandran, Professor Ling Liu, Dr. Omesh Tickoo, and Dr. Padmanabhan Pillai, for their positive and constructive feedback on my thesis. Adhyas Avasthi, my mentor at Nokia Research Center Palo Alto, has been never-ending a source of the ideas presented in this thesis.

Min Lee, his kindness will not be forgotten. Sangmin Park has always shared my problems and given me good workable advice since I first met him. Hyojun Kim(older one) is one of best friends in my life and the perfect example of "Steady does it." Jungju Oh and I overcame a lot of difficulties together. Younggyun Koh's sense of humor cheered me up whenever I felt blue. Gueyoung Jung gave me an incredible opportunity for my future carrier. Thank you for my friends to share my joys and sorrows together: Hyojoon Kim, Byoungyoung Lee, Changhyun Choi, Junhee Park, Kirak Hong, Moonkyung Ryu, Myoungchul Doo, Sangshin Jung, Wonhee Cho, Joohwan Lee, and Youngjin Jang. My fellow students and research collegues over

the years are also appreciated: Adit Ranadive, Priyanka Tembey, Vishakha Gupta, Qingyang Wang, Dulloor Rao, Hrishikesh Amur, Vishal Gupta, Chengwei Wang, Fang Zheng, Liting Hu, Jay Lofstead, Hobin Yoon, Alex Merritt, Sudarsun Kannan, Ketan Bhardwaj, Dipanjan Sengupta and Junwei Li. It was my pleasure to work with Hyunjong Joseph Lee for our recent collaboration. There are tons of other friends who are together with me having a truly great experience in Atlanta. I'm thinking of every one of you as I write this and I thank each of you very much for being a part of my life through my years at Georgia Tech.

Finally, I thank my family, Yoonhwa, Seyoon Selena, and Chaekyung Oliva, for their understanding, love and support through this long journey of my doctoral degree, which would not be possible without their devoted support and sacrifice. Yoonhwa, I don't know how I can ever repay you. Also, I would like to thank the full material and emotional support from my parent and parent-in-law.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Client devices operating at the edges on the Internet, in homes, cars, offices, and elsewhere, are highly heterogeneous in terms of their hardware configurations, form factors, and capabilities, ranging from small sensors to wearable and mobile devices, to stationary ones like smart TVs and desktop machines. With recent and future advances in wireless networking allowing all such devices to interact with each other and with the cloud, it becomes possible to combine and augment the capabilities of individual devices via services running at the edge – in edge clouds – and/or via services running in remote datacenters.

The VIRTUAL PLATFORM approach to combining and enhancing such devices makes possible the creation of innovative end user services, using low-latency communications with nearby devices to create for each end user exactly the platform needed for current tasks, guided by permissions and policies controlled by its access control leveraging user context and social network services. To end users, virtual platforms operate beyond the limitations of individual devices, as natural extensions of those devices that offer improved functionality and performance, with ease-of-use provided by cloud-level global context and knowledge.

CHAPTER I

INTRODUCTION

There have been substantial recent changes in the computing environment accessible to end users, driven by always-connected client devices, software services, and cloud computing. Coupled with an ever-growing number of client devices able to connect to the Internet, there is now an improved opportunity for applications to more accurately capture each person's life and provide assistance in daily tasks. Software services like Google Maps are essential for applications on such devices, because they allow applications to analyze captured data to permit them to better understand and predict people's behavior. Cloud computing scales not only their computing capability, but also provides data about past actions and the actions of others. The idea of virtual platforms (VPs) explored in the thesis proposes a new execution environment that offers clean, high-level abstractions for such devices, services, and the cloud, so that applications leverage them in a consistent and transparent way.

1.1 Motivation

Today's client devices are highly diverse in terms of their hardware configurations, form factors, and capabilities, ranging from wearable devices and smartphones to the stationary ones at home, for example, large-screen TVs and home PCs equipped with convenient keyboards. In addition to the availability of an unprecedented number of such devices, there is an ever increasing number of sensors [11] that can substantially enrich interactions of people with their devices and environments. Recent high-end smartphones, for instance, have more than 10 embedded sensors, and there already are 6 sensors on average in roughly 30 to 40% of today's mobile phones [14, 24], and a similar trend is seen for emergent wearable devices. Similarly and mirroring

the growth in device-embedded sensors, there is an increasing presence of sensors in users' external environments, ranging from those in controlled settings like homes, cars, or hospitals, to those in public settings like sports venues, entertainment, or parks. Along with the advent of such devices and sensors, advances in network technology, including high-speed wireless and mobile communications driven by low hardware costs, have facilitated the underlying trend for all of those devices to become network-enabled[49]. Increased levels of device connectivity – to each other and to the Internet – cause software services like social network services (SNS) to play an increasingly important role in capturing a user's current relationships with people and the world. For example, SNS has become a medium via which users pursue common interests across those in *Friend* relations, by sharing ideas, activities, and events. Finally, popularized cloud services are potentially able to augment devices' innate capabilities such as their storage size and compute abilities [76] as well as those for the software services. At the same time, the cloud has become a medium not only for enhancing individual devices' capabilities, but for offering better personalized services to users via accumulating and analyzing device-collected data like Apple's Siri.

Such progress in device and services backed by network connectivity promotes a new trend in applications, which is the movement from a traditional model in which each application engages only with some single device currently being used to new models in which applications take full advantage of distributed resources across dynamic sets of devices and services, no matter who owns them. This model brings potential opportunities for applications to interact with and assist users in their daily lives: to help capture user intent and context [36], to drive intuitive and meaningful user interactions, and to assist with specific tasks as in elder care [111], efficient driving [118], etc. Those applications, however, face challenges when seeking to leverage and use the dynamic sets of sensors and resources present on client devices and clouds, since all of the underlying resources and services tend to operate in a

highly isolated and fragmented manner.

This thesis explores new ways for applications to exploit the capabilities of **virtual platforms** at the edge of network comprised both of many nearby devices and the remote cloud. Virtual platforms are constructed automatically, where SNS and current user context are used to guide their construction and use (i.e., permissions and policies) for participating end users. Virtual platforms create the resource pools that deliver capabilities to the applications that require them, and access controls concerning the use of such capabilities are enforced via access control policies. To illustrate, consider a user who wants to edit video clips via some application installed in her smartphone. The single device model constrains the application to use the smartphone's processor, memory, screen, and storage, whereas device couplings allows it to use, for example, the processor of the high-end home server that supports a special instruction set to process multimedia data, the large storage on the cloud, and the TV as a display screen, as well as other nearby – neighborhood – devices based on existing SNS relationships. Such couplings, therefore, make it possible for the application to acquire for its use the 'best-fit' capabilities from distributed nearby and cloud resources, to run with the fidelity, delay, and throughput desired by end users.

A desirable property of virtual platform for applications is to have uniform access to underlying resources, whether those are provided by home devices or remote/edge cloud components. Further and essential to running applications across such resources are mechanisms that manage devices and cloud services as members of existing couplings, as well as an organized way to control resource access. The aforementioned use of SNS provides such functionality. The goal of virtual platforms is to propose a high-level abstraction and execution environment combining all of accessible resources, services, and data beyond the boundary of a single and individual user and device.

1.2 Thesis Statement and Contributions

This thesis develops and explores system-level support for running end user applications across pools of resources present in users' homes, neighborhoods, and other nearby facilities, as well as in the remote datacenters provisioning cloud service. The approach maintains these pools and uses them to dynamically compose and maintain for each application the set of resources that match its current needs. Termed a *Virtual Platform* (VP), its resources are uniformly accessible to the application, used as if they were present in an imaginary single device, but are actually internally comprised of local, nearby, and remote cloud resources, with a novel access control mechanism.

The thesis claims the following key advantage for virtual platforms compared to isolated devices: *by creating a virtual platform as an abstraction consisting of uniformly accessible local and remote resources, and by using current user context and social network services for access control, applications can obtain higher levels of performance, improved ease-of-use, and improved fidelity, going beyond the limitations of individual devices.*

The approach chosen to provide uniform access is for the system infrastructure to represent and run applications as sets of computational, sensing, actuation, user interaction, and storage services, with individual services mapped to the resources where they run *best*. An example is a service using a camera device on a child's phone, with display and interaction services running on the mobile device currently used by the parent, perhaps along with a storage device in the cloud and with support services that perform data routing and interpretation. Permissions to run in this fashion would be provided as long as the parent and the child maintain *friend* relationships on a jointly used SNS.

Specific contributions of this thesis include the followings:

- VP software manages and dynamically composes distributed networked resources that are from both local/personal and remote/public devices and machines, where any of those entities can be active participants in running the services desired by end users, and where all resources in the infrastructure are available to the applications being run.
- Participation in a VP is guided by permissions and policies controlled through a novel access control service leveraging current user context and social network services (SNS), thus making it possible to share devices owned by different end users and/or residing at different locations. This is done in a privacy-preserving manner. With the access control service, the infrastructure can extend and alter a VP without the need for direct and repeated user interaction or consultation.
- VP software tracks the availability of networked resources and/or decides what resources should be used by some VP instance to meet its current needs. Allocations are guided by policies aware of current device capabilities and network conditions.

Figure 1 suggests the idea of virtual platforms as a new execution environment integrated diverse distributed resources for the applications.

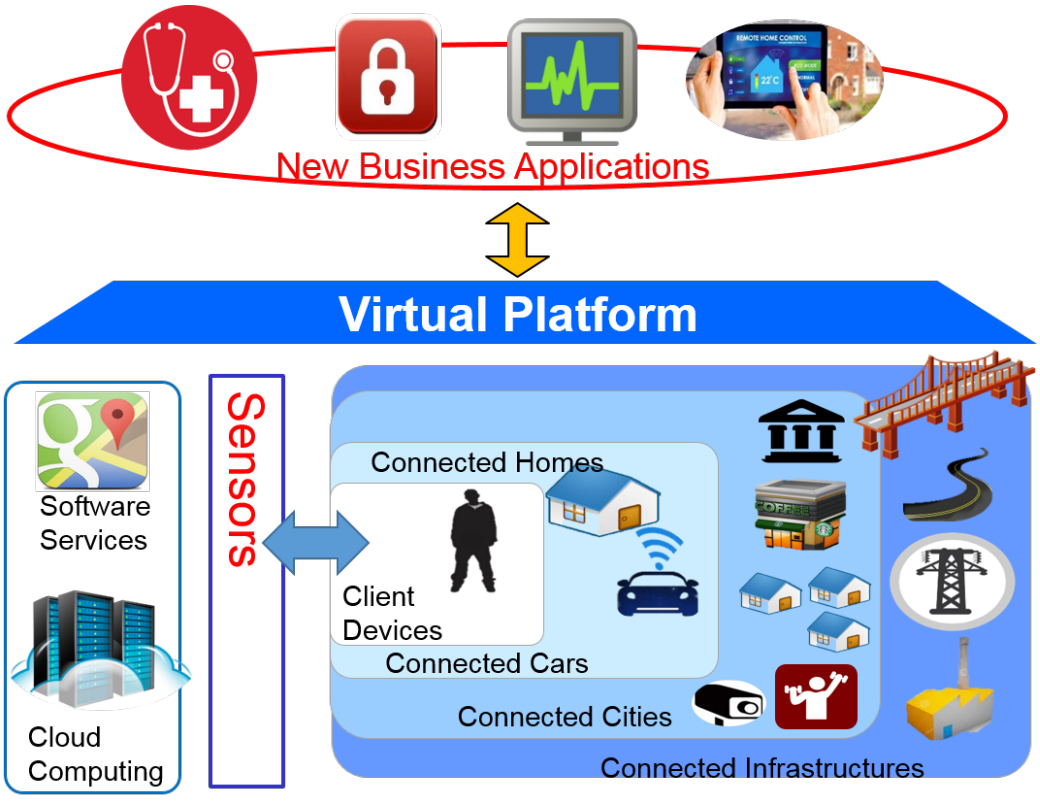


Figure 1: The virtual platform as a new execution environment.

CHAPTER II

BACKGROUND

There is an increasing number of devices that users interact with in homes and offices, driven by continued improvements in device capabilities and network connectivity. While these devices are capable of interoperating with each other to enhance applications' user experiences with regard to their hardware capabilities, current systems and software remain lacking in terms of their ability to freely combine their capabilities to provide the most current value to applications serving end users. For instance, it should be easy for applications and users to take media being played out on a mobile device and display it on a homes large-screen TV as soon as the person using the device enters her home. It should be similarly easy to evict power-hungry tasks or applications on the battery-operated mobile devices and instead, move them to a desktop at home.

Our initial explorations concerning applications' ability to run across multiple devices centered on a proof-of-concept system, the STRATUS infrastructure [66], which dynamically constructs – synthesizes – virtual platforms from the devices present in end user environments like homes or offices. A virtual platform constructed by STRATUS is an abstraction of a single logical device consisting of resources from diverse devices over the network. This chapter highlights the design principles and technical challenges of the idea of virtual platforms, addressing issues like device interoperability and other topics relevant to the VP concept.

2.1 Stratus

With the STRATUS infrastructure, the goal of virtual platforms is to enhance applications' users experiences across multiple devices in end user environments. Recent

studies [50, 93, 91] show that users prefer seamless and integrated interaction of those devices rather than leaving them as many individual, isolated, and independent ones. Such seamless integration is easily realized by STRATUS , as it creates a VP as a single abstraction across multiple devices on the network. A STRATUS VP allows the applications to easily and transparently span multiple end user devices.

2.1.1 Assumptions

As a proof-of-concept toward seamless device integration through VPs, STRATUS prioritizes design space exploration to identify the required functionality of a virtual platform, in particular focusing on design decisions concerning the underlying system used to construct a VP. STRATUS limits its support to existing applications rather than proposing new SDK for future applications interacting with a VP. Along with this limitation, STRATUS makes the following assumptions:

- STRATUS follows the current practice for end user applications on personal devices, which is that they assume the use of a single computing device [51]. So, STRATUS provides to applications an abstraction of a single device that actually consists of multiple, underlying networked devices. Moreover, since users expect the same interaction experience across all of their devices [93, 91], STRATUS preserves the same computing environment across a user's multiple devices, shielding the applications from modification to run on a VP.
- To leverage the physical proximity of devices on the network, STRATUS assumes that all participating devices for a given VP are on the same local network with one-hop distance. This permits a VP to be more responsive to interaction with applications [64].
- Since all devices for a virtual platform should be trustworthy, we assume single ownership of the devices [93] being used.

2.1.2 Architectural Design

Current infrastructures do not offer the flexibility in device use and seamlessness of operation required to easily realize the usage examples introduced earlier. Our novel approach to obtaining these properties is to implement cooperative behaviors via the virtual platform abstraction realized by the STRATUS infrastructure. The purpose of cooperation is to exploit the heterogeneity of the resources present in such Stratus-enabled systems, to provide to end users exactly the virtual platforms they desire.

2.1.2.1 Design Principles

Virtual platforms in STRATUS seek to offer better and/or more efficient resources to applications than what can be provided by single physical devices. To reach this goal, STRATUS permits each device to export its own diverse and heterogeneous resources, logically decoupling those from the device. An example is a PC divided into a set of resources including its computation, memory, storage, input and output resource, where STRATUS uses each resource over the network to construct an abstraction of a single device, a virtual platform, needed by some end user. This decoupling contributes to STRATUS hiding the low-level details about such networked resources (e.g., connection protocols like WiFi, Zigbee, and uPnP). STRATUS never assumes that a device has a pre-determined single role when it constructs a VP. Rather, it leverages individual resources extracted from a device. In addition to its fine-grained resource control, an additional principle is that users and applications should be able to inform the STRATUS infrastructure about their respective needs, in response to which STRATUS constructs a VP and operates resources that meets those needs. Moreover, STRATUS actions are also guided by system-level requirements to better operate underlying resources, an example being controlling the energy consumption of all devices, while it scales up and down a set of resources. Yet throughout, STRATUS

provides interfaces for each VP that permit existing applications to transparently access local and remote resources without requiring them to be reprogrammed. To do this, we devise a mechanism for STRATUS (i) to realize such interfaces, and (ii) to connect them to the right resources. Lastly, STRATUS maintains the same computing environment across all devices running diverse software and hardware.

2.1.2.2 Required Functionality

The following functionality realizes the aforementioned principles: (i) *Operation of a Collection of Devices*: methods and mechanisms for runtime device discovery, inclusion, and exclusion in virtual platforms, thus relieving applications from these tasks, (ii) *Fine-grained Resource Control and VP Management*: software for dynamic platform construction and configuration, including to monitor the current properties and capabilities of participating devices and thus, the composite properties of the virtual platforms into which they are assembled, and (iii) *APIs and methods for accepting applications and system needs*: executable encoding of applications and system needs to drive (i) and (ii). Additional useful functionality to be explored in future work includes authorization and rights management, in order to deal with shared devices that may not be wholly owned by a certain user, the secure and reliable inclusion of open facilities like those offered by public clouds, and multi-site device clouds like those spanning multiple homes or offices. Such functions were not part of the STRATUS infrastructure, its implementation assuming that all participating devices are trustworthy, that network delays to be negligible, and that there is sufficiently high intra-cloud network bandwidth, as described in Section 2.1.1.

2.1.2.3 Key Concepts

STRATUS introduces the three key concepts – Virtual Platforms, Device Cloud, and Policies – to realize the functionality discussed above. As the VP concept is discussed earlier, we next talk about device clouds and policies.

Device Clouds: To realize virtual platforms, STRATUS maintains a resource pool, a *device cloud*, consisting of virtualized resources logically decoupled from a single physical device. We use the term device cloud to emphasize the fact that any device attached to or part of a Stratus-enabled device can participate in a virtual platform, as long as the device itself is dynamically discoverable and shareable. Inherent properties of device clouds are: (i) each cloud is comprised of a set of devices participating [94] in some common and dynamically synthesized virtual platform, (ii) users have a seamless experience across multiple devices [92, 112] participating in a device cloud, (iii) and, as discussed earlier, the participating entities need not be entire devices, but may provide only some of device components as the resources with which to construct virtual platforms, like computation or storage resources.

Policies: STRATUS maintains device clouds and virtual platforms in accordance with two policies. *Management policies* enforce desired global properties for a device cloud, while *application policies* act on behalf of certain classes of applications [113] or for certain workload characteristics [102]. Virtual platforms are automatically constructed based on application policies and device clouds are continuously managed by management policies. The performance policy is a sample management policy that forces all devices that host VPs to maximize computation performance regardless of their power consumption. The media policy is an application policy demanding that a virtual platform guarantees media files to be played without dropping frames. STRATUS policies differ from prior work on SLA- or SLO-driven system management [79] in that they are used both for constructing and for managing platforms, with different policies encoding different construction and management requirements.

In short, STRATUS enhances end users' experiences with their applications since virtual platforms combine a best set of resources in device clouds to meet application policies that can deliver performance or functionality beyond what is available on a single physical device.

2.1.3 Stratus Realization

System virtualization makes it easy to realize the functionality required by STRATUS . This section briefly discusses the virtues of system virtualization for building Stratus-like infrastructures.

2.1.3.1 Using System Virtualization

STRATUS uses system virtualization to operate a device cloud and manage virtual platforms at hypervisor level, with its current implementation using the Xen open source hypervisor [32]. STRATUS extends the concept of traditional virtual machines(VMs) such that hypervisors can interact with networked resources beyond a single device boundary as well as local resources. A special communication channel [81] in STRATUS enables this transparent remote access for hypervisors, Xen for the STRATUS infrastructure, to handle remote resources in the exactly the same way as for local ones, so that hypervisors are capable of virtualizing remote resources without additional support. In doing so, STRATUS can exploit the advantages of system virtualization, including:

- fully support existing applications, by offering them an abstraction of a single device,
- the same system interfaces to resources, local as well as remote [78, 77], without any dependence on vendor-specific protocols or hardware features,
- the same computing experience across all devices, by encapsulating users work environment in a single VM and using VM migration, and
- easy disaggregation of fine-grained virtualized resources from a single physical device.

In addition, recent study [64] shows that system virtualization can be more flexible than the process virtualization [90, 108] and more general than language-level virtualization (e.g., Java and .NET framework) or composition [100] when realizing a

mechanism for device interoperability.

2.1.3.2 Policy Operations

STRATUS ' use of system virtualization ensures that virtual platforms are elastic in terms of their capabilities and performance, where elasticity is obtained through modifying the mappings from virtual to physical resources as well as by using virtual machine migration. Such elasticity is driven by desires expressed in policies. These determine the resources that a virtual platform should include (e.g., the policy for processing HD video clips is designed to establish a virtual platform that includes a HD accelerator and a large screen), and they can express additional performance or power attributes (e.g., the platform should use minimal amounts of energy consumption). STRATUS uses these policies along with its information about current devices and their resource availability to construct the right VPs and to dynamically configure them when devices arrive or leave or when requirements change.

2.1.4 Implementation Details and Experimental Evaluations

For implementation details and experimental evaluations of the STRATUS infrastructure, readers are encouraged to refer to the full paper [68]. We next summarize the key lessons learned from STRATUS , as background to the work presented in the remainder of this thesis.

2.1.5 Discussions

We developed a proof-of-concept system, STRATUS along with VPs to obtain device interoperability and applications' user experience improvement. Its results [68] indicate that virtual platforms derived from device clouds are able to successfully deliver the functionality in need. Advantages include higher performance, the same computing environment across multiple devices, and improved end user experiences. These properties are due to the elastic and dynamic nature of virtual platforms, driven by

policies encoding end user needs.

The STRATUS realization of device clouds is implemented with the Xen hypervisor, which allows STRATUS to simplify fined-grained resources creation and control with the existing interfaces and migration of user's computing environment across devices. Although virtualization makes it easy to implement such functionality, it also imposes on STRATUS the severe limitation that non-virtualized devices and online services cannot be a part of a device cloud. This limits the generality of the approach, particularly in lieu of the fact that most mobile devices are not virtualized.

Furthermore, STRATUS uses VM migration to preserve the same computing environment across multiple devices. In theory, the migration should work as far as devices have the same hypervisor, but in reality, migration between different architecture like ARM to x86 is difficult. It is also likely to cause heavy overhead for battery-operated devices in terms of resource use and time taken for migration [83, 90], so a lighter weight mechanism is desirable.

Elasticity for VPs may be limited by the strictly local scopes of individual devices, i.e., VPs only use the resources existing in the current device cloud consisting of locally available devices (see our assumptions in Section 2.1.1). The STRATUS implementation simply rejects VP creation when resources in a certain device cloud are insufficient for meeting application needs, even if additional flexible resources are available in remote clouds and/or in ambient devices. This indicates the need for implementation methods that can bring such cloud and ambient resources to device clouds, in a safe and secure manner.

2.2 Related Work

2.2.1 Providing seamless user experience based on virtualization

Prior work has used virtualization technologies to provide users with seamless computing environments. ISR [74], the Collective project [41], SoulPad [40], Keychain [30],

Spirits [98], and Cloudlets [107] store all of a user's computing environment and state in virtual machines, using VM migration to move VMs to the device the user wants to use. VMs can be migrated through a network connection, via mobile storage devices like USB disks, and they may be assisted by nearby resources so as to avoid the potential high overheads arising from accessing remote cloud resources [98, 107]. All of these approaches target single physical platforms, however, in contrast to the STRATUS approach in which multiple platforms' devices are utilized to gain improved end user experiences. Also missing from such work is the automation via policies shown in our work.

2.2.2 Offloading the compute-intensive tasks of applications

Prior work has leveraged remote computing resources to augment the performance of devices. Similar to the STRATUS infrastructure, Cloudlets use system virtualization to migrate a user's VM on a mobile device to a nearby small server that offers augmented computing performance. While Cloudlets only deal with the compute capabilities applications, STRATUS combines heterogeneous resources including hardware accelerators and various I/O devices on the networked devices, to improve end user experiences. There are also middleware frameworks for offloading compute intensive tasks to resource-rich servers. Recent research like MAUI [47], CloneCloud [43], Cyber foraging [31], and Cuckoo [71] address the issue of how to find the compute intensive portions of an application and move those between battery-operated mobile devices and resource-rich servers. They focus on remote cloud servers, so these systems do not have functionality that leverages the various capabilities present in the many and often heterogeneous devices shown in the STRATUS infrastructure. There is no clear distinction between nearby and remote cloud resources in such research, which is essential for interactive application response [107]. Like Cloudlets, their focus is on compute-intensive tasks. Finally, since they require developers to use their

own SDKs or special annotations to identify where to offload, existing applications cannot easily take advantage of their capabilities.

2.2.3 Thin-Client Computing

Building on previous work like thin-client [21] computing, an alternative model offered by cloud computing is to rely on remote server systems to provide needed functionality [80, 3, 2, 23]. In this approach, user tasks and/or applications are always hosted by a remote 'home' server, over the network. The distance between a client and the server varies from one-hop to far-remote, which possibly gives rise to latency issues. Device clouds simply extend those models to also exploit locally available devices, in addition to using remote services. This may also help deal with the privacy and security issued raised for cloud systems as well as improve responsiveness, by differentially using local vs. remote service capabilities [119, 22, 6].

2.2.4 Composition of Software Modules

Technologies like CORBA, DCOM, .NET Remoting, and Java RMI help developers compose software modules over the network or in a single machine when writing an application. With such middleware, complexities regarding handling different protocols and diverse devices are left to each application. Stratus hides such complexity from applications through providing the interfaces that they already use. Unlike the Stratus approach, they require existing applications to be reprogrammed in their SDK. Also, they lack the ability to manage the resources for each service on a node (e.g., scale up, down or dynamic resource allocation). However, they would provide a discovery and composition mechanism to VPs since Stratus is agnostic to such mechanisms.

2.2.5 Context-related device interoperability

Many systems such as Gaia [103], BEACH [35] have been proposed to coordinate software modules and diverse networked devices contained in some physical space. Such systems assume a fixed role for each individual device at a fixed space. STRATUS makes no such assumptions. HomeOS [51] is a management framework for devices at home, offering access controls with device interoperability via a PC-like abstraction. To hide complexity when applications access distributed resources, SpeakEasy [52] proposes a device-to-device interoperability protocol, and uMiddle [89] provide an interoperability layer in the infrastructure not each edge. None provide a set of fine-grained resources from a single device, as supported by STRATUS .

2.3 New Design Principles for Future VPs

STRATUS provides insights into the design and implementation of virtual platforms created to enhance end user experience 2.1.5, along with the prior work presented in Section 2.2.

STRATUS identifies the following fundamental features as useful for realizing the idea of virtual platforms: (i) its mechanism for fine-grained resource control and resource pool management and (ii) its ability to offer an abstraction of a single device so that applications can transparently operate across diverse resources over the network. We argues that a future infrastructure for VPs should maintain those advantages, but also assert the need for additional functionality to handle the cloud and ambient resources, beyond single-person ownership. Specifically, while STRATUS VPs support existing applications, their consequent lack of awareness about the device-rich environments in which they operate forego opportunities to explore new kinds of applications and services. Related work has explored alternative approaches to enabling novel applications and software services, by composing new applications from software modules or objects rather than entire VMs. This allows each application,

for instance, to opportunistically offload computation-intensive tasks [47, 43, 73]. It also helps non-virtualized resources participate in a device cloud. In such settings, the idea of policies should be evolved to capture an application's 'intent' regarding resource composition [36], directly provided by an application, with the assumption that applications have clear goals concerning resource use [36, 95, 122]. An example is the manifest description in HomeOS. The disadvantage, of course, is that developers need to use a new SDK, leading to potential lack of acceptance.

Another generalization of STRATUS is one that encourages more devices to make themselves available for participation in virtual platforms. From the VP point of view, the more devices join a device cloud infrastructure, the better virtual platforms can be created because of richer resource pools and consequently increased elasticity. Encouraging device and resource sharing can assist in that task, but also raises security and privacy concerns that need to be addressed.

In summary, the following insights derived from STRATUS guide our future design:

- An abstraction of a single device with a fine-grained local and remote resources,
- An enhanced infrastructure to enable new applications maximizing the device-rich environments that devices operate, and
- A novel mechanism to encourage more devices to participate in virtual platforms in a secure and safe way.

CHAPTER III

PERSONAL CLOUDS: SHARING AND INTEGRATING DEVICES TO ENHANCE END USER EXPERIENCES

This chapter extends the idea of device clouds and virtual platforms presented earlier to propose the Personal Cloud infrastructure reflecting new design principles to evolve the idea of virtual platforms based on what we learned from the Stratus infrastructure. The virtual platform with the Personal Cloud infrastructure creates a Personal Cloud (PCloud) instance for each application to transparently interact with resources on a device or over the network. Unlike Stratus, the PCloud infrastructure can transparently leverage resources that are not owned by the current user due to its access control service, and helps the applications improve their scope of data beyond a single device.

3.1 Goal

End user experiences on mobile devices with their rich sets of sensors are constrained by limited *battery life* on devices and restricted *form factors*, as well as by the *scope* of the data available locally. The goal of the Personal Cloud abstractions is to address these issues by enhancing the capabilities of a mobile device via seamless use of local, nearby and remote cloud resources. Such devices like smartphones are operating in increasingly rich settings that include not only nearby sensors and machines, but also the remote cloud. Hence, by leveraging and interacting with such potentially cooperative resources, the capabilities of the devices can dramatically be improved, and device users can gain enhanced interactions with their current environments [43, 44, 47, 106]. Another goal is to leverage resources beyond individual user's ownership

when offering a PCloud instance to the applications, which allows them to maximize the use of all available resources in their environment.

3.2 Contributions

The idea behind Personal Clouds (PCloud) is similar in spirit to vendor-specific or industrial solutions for integrated use of shared devices, such as Apple's AirPlay and Microsoft's Smart Glass as well as DLNA. However, in contrast to those solutions operating only across *compatible* vendor-certified entities, PClouds have no such constraint, by using a simple model of device interaction realized at a level of abstraction *below* that of vendor-specific software, i.e., at the system level. Specifically, the Cirrostratus implementation of PClouds operates as a set of extensions of the Stratus infrastructure in the Xen hypervisor [33], and it interacts with non-virtualized, i.e., devices without supporting virtualization, like Android-based smart phones, via additional device-resident *agents*.

Generalizing earlier work on *device clouds* [66], Personal Clouds make the following new technical contributions:

- They manage and dynamically compose distributed networked resources that are from both local/personal and remote/public devices and machines, where any of those entities can be active participants in running the services desired by end users, and where all resources in the PCloud are available to the applications being run. The digital neighborhood watch (DNW) application [38, 42] with a PCloud instance, for instance, takes advantage of PCloud to run its face recognition service both with and without remote cloud connectivity, albeit at different levels of fidelity based on where and how it runs.
- Device participation in any PCloud is guided by permissions and policies controlled through social network services (SNS), thus making it possible to share devices owned by different end users and/or residing at different locations. This

is done in a privacy-preserving manner, via a system level service for authentication and authorization that uses *Friend* relationships on Facebook to look up and encode the relationships of users' participating devices. With such SNS-defined access policies, Cirrostratus can extend and alter a user's PCloud without the need for direct and repeated user interaction or consultation.

- PClouds protect end user privacy by tagging data – like photos – with metadata about the devices on which it is captured, the users to which it belongs, and other such semantic information. This makes it possible to automatically *sync* data across of all of the devices owned by some PCloud user, e.g., with a privacy-protected repository maintained on her home desktop vs. storing such data in some remote SNS not controlled by her. The user can then selectively upload photos from the repository to the SNS, and/or she can use a PCloud-provided service that gathers photos from the SNS, e.g., from those who are encoded as *friends* and have expressed their intent to share their pictures on Facebook.
- The PCloud runtime tracks the availability of networked resources and/or decides what resource should be granted to a PCloud instance to meet current demand. This is managed by Cirrostratus with the awareness of current device capabilities and network conditions. An example is a face recognition service located on three different platforms (e.g., a mobile device, a nearby home desktop, and a remote cloud) with different choices determined by current network connectivity and desktop load providing different levels of performance to PCloud users.

3.3 Motivating Use Cases

This section demonstrates how Personal Clouds can leverage surrounding and cloud resources to provide end users with enhanced application functionality and performance.

The first use case implements digital neighborhood-watch functions. It uses face recognition to distinguish neighbors from friends from others, where Cirrostratus orchestrates its use of network-accessible resources in the home and/or on remote cloud machines. Using local resources offers rapid response, but may suffer in terms of accuracy due to limited volumes of face data on the home machine.

Additional use of remote machines improves accuracy by drawing on the cloud's global scope and data. The second use case shares devices and content among friends, where the SNS is used to establish secure interactions across friends' devices. The applications enabled by this functionality include playing a video on say, a friend's large-screen display, capturing sound from other devices, and playing sound where desired.

3.3.1 Digital Neighborhood Watch

A neighborhood watch is a community group organized by residents to forestall crimes and vandalism. Tasks include watching for suspicious activities and persons, notifying each other of such events, alerting neighbors of other neighborhood issues, and reporting select events to external parties like the police. In this context, a face recognition service can be used to classify persons seen in the neighborhood as residents vs. others, e.g., using images taken by neighborhood or home cameras and stored in homes' image repositories. Such a service should be always-on, even when the Internet connectivity is not present or when power is out, the latter requiring its ability to run on single battery-driven devices. At the same time, face recognition accuracy heavily depends on the size of the data-set being used [62], with small

sample data for each face, i.e., a narrow *scope* of data, resulting in over-fitting and making recognition unlikely. Also important is the computational capacity available for running these codes, with their compute requirements directly related to database size. Further, when images are seen, it must detect and identify potential threats in a timely manner. This makes the network bandwidth and latency between cameras and computing resources running the service critically important. PClouds provide the functionality needed for flexibly running neighborhood watch services. First, the face recognition service can, of course, run on a single device, using the photos resident on it, but second, it can obtain a wider scope of data (e.g., people's faces in this case) than those existing on a single user's device by also accessing a user's home image repository, as well as using remote data available via SNS (or even in neighbors' home repositories). Third, the service can benefit from both home machines' and remote cloud resources' computational capacities depending on local vs. remote network conditions. Section 3.6 describes additional implementation detail about how PClouds backed by Cirrostratus permit the face recognition service to run with these diverse configurations.

3.3.2 Display Sharing among Friends

With high quality video clips and photos with multi-million pixels, recent smart phones have become equipped with high resolution displays and high performance processors. Small screen sizes continue to hamper user experiences, however. PClouds permit mobile devices seamless access to nearby large screen displays. Our previous work demonstrated this capability [66], but it required all personal cloud devices (e.g., display, keyboard, and processors) to be virtualized and owned by the same user. PClouds extend device sharing to also include non-virtualized devices and those owned by friends. Specifically, users can share any capability of any nearby device (e.g., a large display in this case), enabled by the use of social network services

(SNS): the Cirrostratus infrastructure implementing this functionality ensures (i) that the device issuing a sharing request is actually owned by a designated friend who is identified via SNS, (ii) that the request recipient is the intended PCloud, and (iii) that no one else is currently using the chosen target device (i.e., the display). Secure, access-controlled sharing is realized with a security service implementing the X.509 specification and using the SNS to maintain *friend* relationships; the devices currently supported include user's home displays, storage devices, keyboards, and machines, devices in friends' homes (even image repositories they wish to export), and remote cloud resources.

3.4 Approach

Personal Cloud instances are created at the hypervisor-level by the federation of networked resources, which is best suited for the current application's demand. In addition to its deployment at hypervisor-level, it integrates nearby and remote cloud resources in an uniform way that the applications are able to transparently access any resources regardless of their locations, i.e., the local device, nearby devices, and remote clouds. This attribute permits PClouds to service end users even when remote cloud resources are not present and/or difficult to access due to insufficient network connectivity or when they are expensive to use via metered connections like the 3G/4G mobile wireless network. This is because a PCloud can also run on available and free-of-charge user-owned resources in the environments such as the home or offices. To this end, the Cirrostratus infrastructure can federate a user's networked resources to establish a personal execution environment for the applications, governed by policies that go beyond evaluating network connectivity to also consider device ownership and access rights, the latter managed in a secure fashion via standard Social Network Services.

3.4.1 Advantages of Personal Cloud Approach

The novel approach in PClouds allows applications to obtain substantial benefits from using local, user-owned nearby vs. remote cloud resources, in part due to the low latency of access to nearby devices and due to improved privacy when using them. The latter is because users retain complete control over their data when storing it locally rather than placing it into the cloud (e.g., voice, pictures, etc.). The outcome is that a PCloud no longer limits mobile applications to run on single devices. Instead, it can exploit the capabilities of the variety of devices available in most homes, offices, and elsewhere, and the power of remote services present in the cloud. Advantages derived from using PClouds compared to single devices include the following:

- **Combined and augmented abilities**. While mobile devices are imbued with many built-in sensors, the interpretation of sensor outputs can benefit from increased computational abilities and from data captured previously and/or stored elsewhere, e.g., in nearby desktop PCs or in the cloud. A classic example is a face recognition application using a camera on a smartphone (i.e., a local sensor) to capture images, but leveraging other network-accessible resources for computationally intensive work and to deal with the fact that recognition accuracy also depends on the extent of the face database. The latter holds because in contrast to storage in a user's home, a cloud service can store both the faces available in the user's local context and those available from the Internet.
- **Improved usability**. While the small form factors of mobile devices may restrict their display and keyboard sizes, this is not the case for the large-display TV in a user's home (or his friend's) and the keyboard attached to his home desktop machine. We demonstrate new and secure methods for accessing and using/sharing such capabilities available on nearby networked devices that are owned by users and/or their friends or other cooperative parties.
- **Increased scope**. Referring to the fact that limited storage size of a device

and current context (e.g., physical location) can benefit from data resident in the remote cloud and captured by other devices, a storage aggregating remote cloud and all user-own devices is shown capable of delivering improved accuracy and utility for services offered to end users.

- **High availability**. Studies[47, 44, 43] show that applications on battery-operated devices can gain performance and availability, and extend their battery lives by offloading computationally expensive tasks from local to remote resources. PCloud can do so by seamlessly tying applications running on battery-operated devices with both nearby and remote resources.

The next section explains how we construct, maintain, and configure personal clouds.

3.5 Architecture Design

PClouds are realized as a layered software architecture designed to operate within dynamically changing external environments. Figure 2 depicts the main components of the proposed architecture.

3.5.1 Applications

An application is a set of services running on a PCloud instance. An example is a media player consisting of a storage service holding a content file, a decoding service generating a video stream from the file, and a screen service projecting the video stream. The PCloud API permits cloud instance creation, query of and access to currently available services, and the construction of new services for a given one. Designed to support existing programing models and minimize code development, a PCloud application operates much like one running on a single device or machine. It runs a simple preamble to create a PCloud instance, then runs its services, the latter able to use all of the resources in PCloud. It is the responsibility of Cirrostratus and its runtime to ensure resource availability in lieu of changing network conditions and

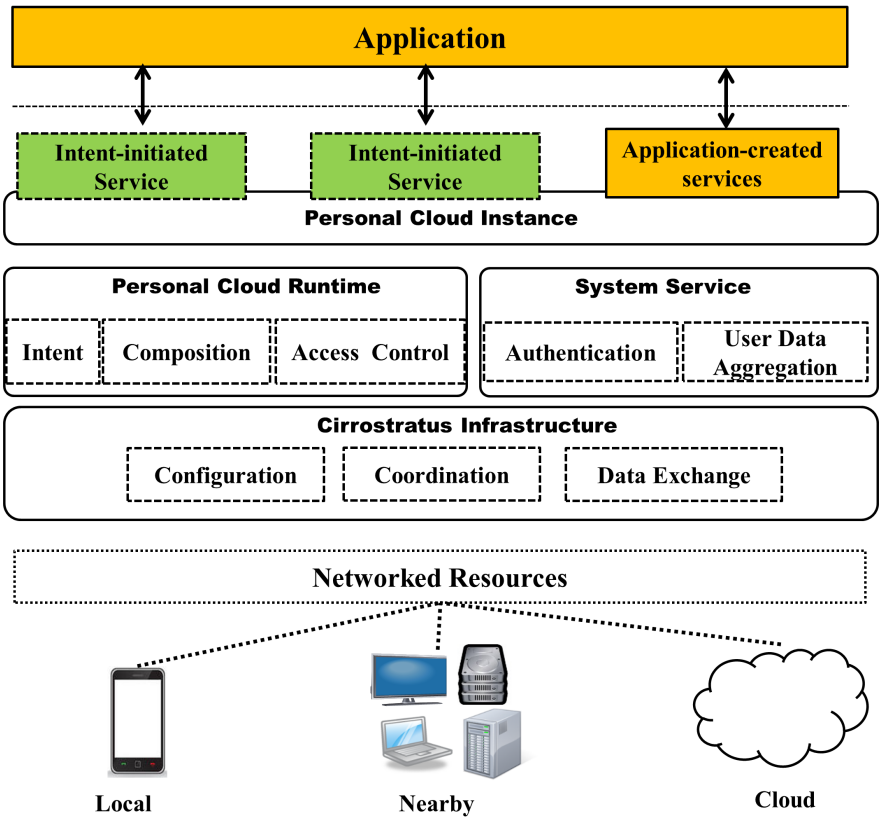


Figure 2: Personal Cloud Architecture

resource use by others.

3.5.2 Services

Applications running in some PCloud instance can transparently access both local and remote services. This approach is similar to that of the Android Service implementation in the sense that it separates computing intensive or hardware access parts of applications from their main execution flows.

PCloud services are like those described in the Intel processing framework [5], but differ in implementation in using local *service proxies* for the services that are run remotely. Specifically, for each remotely run service, the runtime instantiates a local proxy that reveals how to reach the remote service, what service is required for dependency, and what resources are needed to run it. State information is maintained, as well, relayed by the runtime to the actual remote service location. The current API has operations to enumerate, attach, use, and detach services, and to create and launch new ones. System services differ from others in that they are always present and available.

3.5.3 Personal Cloud Instance

Each PCloud presents the illusion of a single machine. Its resources may be distributed across some set of networked devices, but they are accessed in exactly the same fashion as done by traditional applications (e.g., `open`, `read`, and `write` system calls). An instance's life-cycle transitions from its initialized, to started, to terminated states.

The *resources* presented in a PCloud instance are either services or devices, and for both types of participants, Cirrostratus exposes and maintains their capabilities. In terms of the granularity of managned resources, Cirrostratus is very similar with that of Stratus. For instance, a desktop machine can separately expose devices like its disk storage, processing capacity (CPU), and large display. This means that a PCloud

application can run across say, the CPU of a laptop, the storage on a desktop, the keyboard of a smartphone, and a separate large-screen display of a TV, all as if it were running on a single platform offering all of those device resources. Other than running the aforementioned preamble, the use of hypervisor-level technologies in Cirrostratus makes it possible for entirely unmodified single-machine applications to run across such virtual platforms. However, users may wish to impose constraints on the platforms being constructed. For this reason, the PCloud runtime provides additional APIs for an application to guide a construction of such platforms: the **Intent** runtime module and API can be used to submit a list of required services along with their required capabilities. An example is the 're-wiring' of a user's display device, to say, move the depiction of a streaming video currently being viewed by the user on his laptop screen to his large home TV display. A PCloud instance supporting this example may continue to use the exact same laptop resources as before, and simply add the large home TV as a screen resource.

3.5.4 Personal Cloud Runtime

The runtime brings up a PCloud instance for an application, with its *intent* interface used to describe resource requirements as Listing 3.1, e.g., a large screen with HD resolution, presence of a face recognition service, etc. It also allocates resources for the services being asked for a given instance.

The PCloud runtime's tasks are (i) to establish a PCloud instance with services per a user's request or *intent* – the **Intent** module, (ii) to decide which resources are most suitable for running requested services – the **Composition** module, and (iii) to manage the certificate of the device owner for authentication – the **Access Control** module. The Intent module builds a list of services requested by an application and determines the types of resources needed to run them. Based on its inputs, the Composition module interacts with Cirrostratus to allocate resources, the latter

Listing 3.1: An example of application's Intent

```
1  // intent guidelines
2      intent->guideline.topology = LINEAR
3      intent->guideline.traversal = SYNC
4      intent->guideline.tie = NONE
5
6  // initialize callback
7      intent->callback_table.callback = NULL;
8
9  // Initiate and connect a VP
10     intent->comp = &comp[0];
11     intent = intent->next;
12     intent->comp = &comp[1];
13     intent->next = NULL;
14
```

mapping the best-fit resources to requested services, and it establishes a PCloud instance with such resources and services. The Access Control module works with the authentication system service checks access permissions for the resources requested for some specific user.

3.5.5 System Services

System Services are used to authorize applications to run, and for actions requiring global knowledge, such as authentication and a data storage with the global scope, so they directly interact with Cirrostratus.

These provide global information to other PCloud components and to user-created applications and services. Two default current system services are: (i) the authentication service and (ii) the user data aggregation service. They both interact with social network services (SNS) to authenticate the users of resources joining a PCloud platform, permit and deny requests for resources, and they also use the SNS as a repository of user data for the applications running with a PCloud instance. As stated earlier, authentication leverages SNS. Aggregation is a special storage service

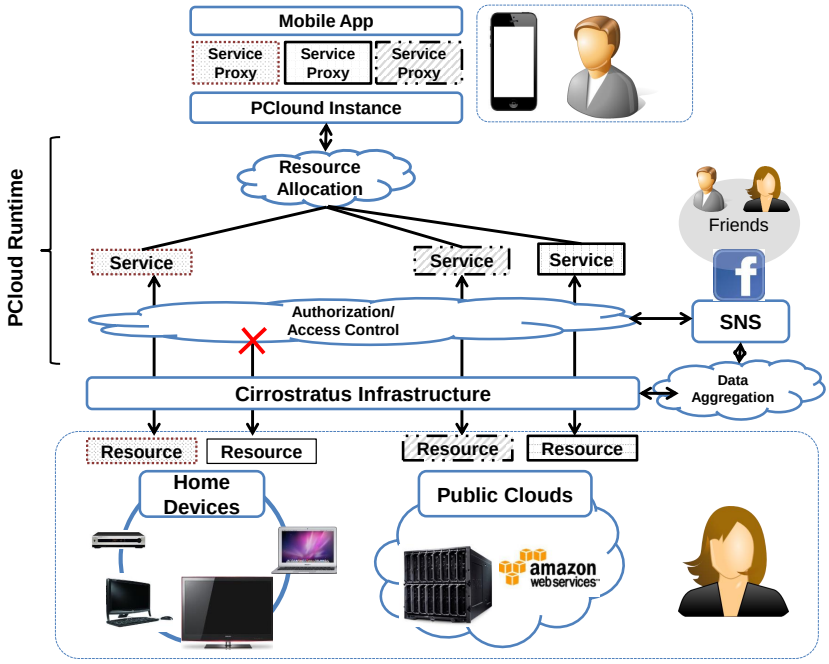


Figure 3: Sharing resources with a SNS-based friend via Pcloud

that assembles data about users from both their local disks and the remote clouds, to create a virtual storage service for each cloud instance. The purpose is for the services running on that cloud to have access to the data they need. Like others, system services also use local proxies.

3.5.6 Cirrostratus

Cirrostratus is an extension of the Stratus infrastructure that discovers and monitors both locally and globally network-reachable resources, in order to maintain a pool of distributed resources accessible to each end user. It also serves as a mechanism layer for the PCloud runtime to draw resources from this pool and combines them to establish PCloud instances.

Cirrostratus extends the earlier work on Stratus and its DEVICE CLOUDS [66]. First, device clouds in Stratus required all active cloud participants to be fully virtualized platforms running the Xen hypervisor. Cirrostratus also permits non-virtualized

devices like Android-based smartphones to be active participants. Second and more important is the use of SNS to implement the access control mechanism needed for sharing resources on devices across *friends* on SNS and for access to friends' data, the latter increasing the data scope available to individual participants. Third, Cirrostratus is able to use remote cloud services (e.g., Amazon EC2) as PCloud resources, thus substantially increasing PCloud functionality and capabilities. Fourth, for all such resources, it is the responsibility of Cirrostratus to maintain status information of about participants' devices, via active device and network monitoring. The **Coordination** module carries out these tasks. Finally, Cirrostratus transport layer, termed **Data Exchange** in Figure 2, is a straightforward extension of the data exchange protocol developed for Stratus. Figure 3 illustrates an example of PCloud use between two users. It illustrates how the aforementioned components relate to others so that users can leverage their own network-reachable resources as well as share others, in a secure fashion, by interacting with a SNS.

3.6 Implementation Detail

This section presents the implementation detail needed for the performance evaluations in Section 3.7.

3.6.1 Cirrostratus

The current implementation supports Linux and Android operating on native or virtualized hardware. We next review interesting aspects of the **Data Exchange**, **Coordination**, and **Composition modules**.

Coordination. Cirrostratus uses the master/slave model to manage the resource pool available for use by PClouds. Its current implementation elects the one with the largest performance index (defined later) as the master among nearby devices. Battery-operated devices are precluded from that role, because of the master's computationally intensive workload and the need for high availability. Slaves contribute

resources with certain capabilities.

Configuration. The configuration module runs on all participating Cirrostratus devices. It uses static data as well as online monitoring to gather information about the capabilities of available device resources, and it provides such information to the master for storage in a central PCloud repository of device capabilities. There are also *knobs* available to device owners permitting them to control their devices' degrees of participation, including preventing certain device resources from being used (e.g., a device's display). Finally, there are resource quotas to control device usage. The naming scheme used in configuration translates `scr1.dt2.pcloud3.bob` to the fact that `Bob` has the `screen1` connected to the `desktop2` in the personal cloud managed by `PCloud3`. A single user may have multiple names, according to his location, say in the home or office. The configuration module also supports resource discovery. That is, if a new device joins any PCloud, this module on the device reports the device's capabilities to the master of that PCloud upon discovery and registration.

Data Exchange. Data is exchanged via an extension of the Stratus channel with publish-subscribe transport [66, 80] built on top of the ZeroMQ library [25] and the Xen event channel mechanism. This channel create a software layer for applications to communicate with others transparent to underlying physical connections being used. Figure 4 shows how it works with distributed resources as well as local ones.

3.6.2 Resource Allocation

Resources must be allocated to services reflecting their demand. The current implementation uses a simple allocation heuristic guided by summary data about resource capabilities changed dynamically. The allocation in PClouds operates within stable allocation epochs, assuming that (i) the status of resources and the network connection are unlikely to change during the current epoch, and (ii) if changes occur,

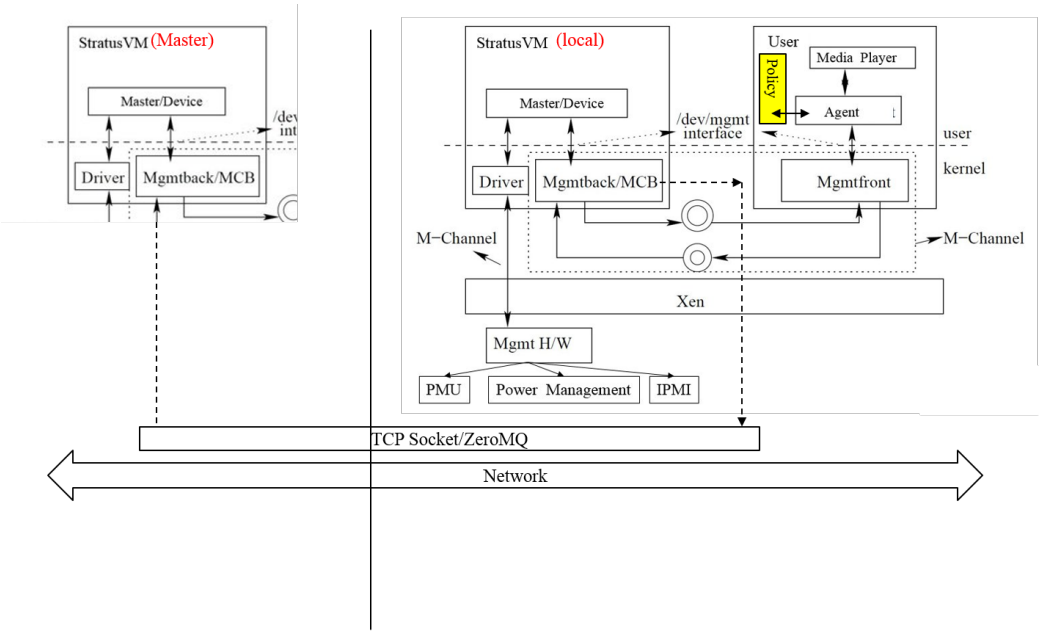


Figure 4: The data exchange implementation in PClouds

they will persist during the next allocation epoch. We have not yet considered multi-epoch allocation planning and/or the evolution of allocations across multiple epochs in place sophisticated allocation methods. We currently approximate the status of each computational resource by computing its performance index (PI) and its network connectivity. The PI is formulated as follows:

$$PI = MemorySize + w \sum_{n=1}^{N} \{freq_n(1 - util_n)\}$$

where

N : the number of cores in a device

w : a weight value assigned to its processor architecture

$freq_n$: the current operating frequency of the n-th core

$util_n$: the average of last five-minute utilization of the n-th core

As the metric for resource allocation, we define the execution time of a service on any resource consisting of computation time and elapsed time over the network as

follows:

$$T_{computation} \simeq \frac{Workload\ size\ being\ processed\ by\ a\ service}{Performance\ Index}$$

$$T_{network} \simeq Elapsed\ time\ to\ transmit\ data\ over\ the\ network$$

Therefore, the estimated execution time is as follows:

$$T_{estimated} = T_{computations} + T_{network}$$

PCloud composition for an application with some set of services, then, operates as follows. When the application requires computation-related services, the composition module interacts with Cirrostratus to obtain the performance index (PI) and network connection status of each candidate computational resource. This results in an initial allocation based only on the PI, i.e., the service is allocated the resources with the largest PI. Runtime monitoring, then, uses the service proxy to obtain additional detail about service execution, including the volume of data transmitted over the network and the elapsed finish time for service requests. Allocations are changed when service times substantially and consistently exceed previously observed values; we have not yet investigated rigorously the dynamic methods needed for runtime re-allocation. Our alternative focus has been on showing PClouds to be viable and useful for realistic application.

3.6.3 System Services

This section describes the system services availabe in PClouds, wich is critical to the ability of PClouds to span distributed resources.

Authentication Service. The authentication service is implemented as a Facebook app, and like other services, its proxy resides within each PCloud instance. A user must install this app on her Facebook account and create a Facebook group named *DeviceShare*. By then listing appropriate Facebook friends, she indicates her willingness to share the capabilities of devices in her own PCloud with others in this group.

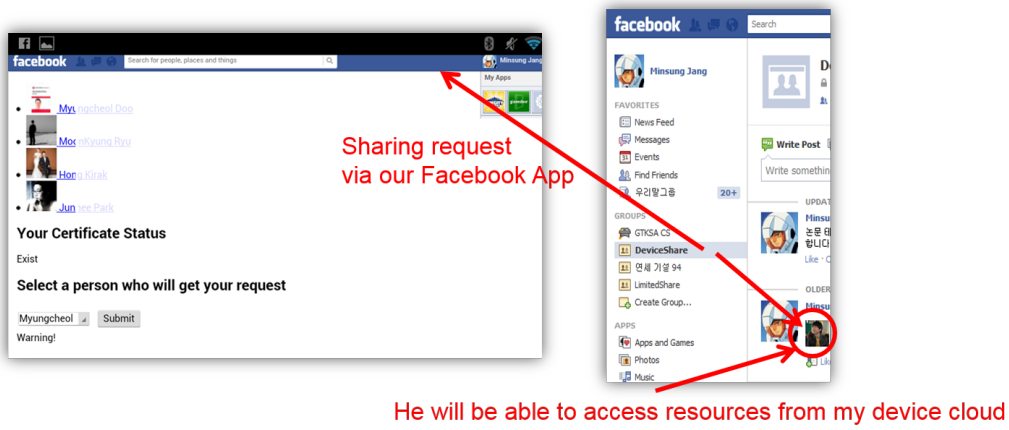


Figure 5: The SNS-inspired authentication in PClouds

Once the app is installed on her Facebook account, the server hosting the app generates and holds her certificate, based on the X. 509 standard. The real working app on Facebook is shown in Figure 5. The following example illustrates the authentication process. Alice and Bob are Facebook friends and install the application on their accounts. One day, Alice visits Bob's house and wants to use his large display to share pictures on her phone with him. When Alice runs the Facebook app (the authentication system service), the app shows the list of people who are willing to share their resources with Alice. Choosing Bob prompts the server to send out her request to his PCloud along with Alice's certificate. The access control module in the runtime then verifies Bob's resources that she is allowed to access. In terms of access rights, the current implementation grants access to resources based on group membership, either owner or guest. In this case, Alice belongs to the guest group, while Bob is in the owner group. Alice's phone also receives the IP address of Bob's network, along with Bob's certificate. With this information, the data exchange module in Bob's Cirrostratus establishes a connection to Alice's phone and then her phone becomes a participant of Bob's Cirrostratus as a guest.

User Data Aggregation Service. The service to aggregate user data consists of the remote and the local aggregated storage. Its remote part extracts photos and

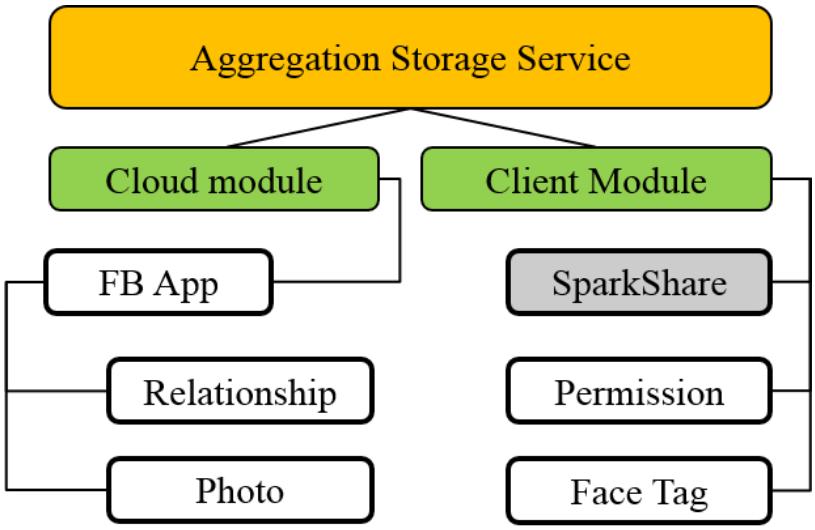


Figure 6: The data aggregation service in PClouds

user-written tags among SNS accounts listed as *friends*, while the local part provides storage to participants of Cirrostratus, e.g., so that it can aggregate all photos from participating devices. The remote is comprised of a Facebook app, which is called the Cloud module in Figure 6, and a remote storage service. If a user installs this app on her account, it collects photos along with tags from friends' accounts that open their photos in albums for friends or the public to see. Since on Facebook, tags of photos usually include information regarding who appears in photos, the app can send both the photos and such meta-data to some remote storage site. For the local synced storage implementation (the Client modules in Figure 6), we use SparkleShare [19].

3.6.4 Virtualized vs. Non-virtualized Devices

Both virtualized and non-virtualized devices in PCloud shares aforementioned implementation detail except the followings.

Virtualized Devices. The control domain of Xen hosts the PCloud runtime and its other necessary components. For both Linux and Android on guest domains, M-Channels link guest actions with PCloud components. We use an Intel Atom mobile platform with the Xen 4.1 hypervisor, running Linux and Android virtual machines.

Non-virtualized Devices. Cirrostratus uses *agents* to interact with non-virtualized mobile devices, such as actual Android phones or tablets, using Google's Android 4.0 Ice Cream Sandwich and 4.1 Jellybean. The agent is a service module on the Java service layer that essentially emulates the functionality resident in the control domain (Dom0) of virtualized systems.

3.7 Experimental Evaluation

This section evaluates the use cases described in Section 3.3 to demonstrate the utility and performance of the Personal Cloud approach and implementation.

3.7.1 Neighborhood Watch with Face Recognition

Test-bed Configuration: We use a resource pool managed by Cirrostratus consisting of two desktops (`N1`, `N2`, respectively), an `m1.large` EC2 instance (`EC2`) as well as a mobile device. Further, the digital neighborhood watch application (DNW) on a mobile device uses a camera connected via Android on our virtualized Intel Atom platform. For non-virtualized devices, the same software uses a camera installed on the Nexus 7 tablet. The face recognition service operates in three different configurations: (i) on a standalone mobile device, (ii) on a PCloud instance with a set of nearby devices, and (iii) simultaneously using a remote server running on the Amazon EC2 instance as well as (ii). Each configuration performs tasks of the DNW, which includes taking a photo, detecting and identifying faces on the photo, returning results to a user, and starting all tasks over again every 3 seconds. In addition, we introduce two different executing environments, exclusive and mixed, in terms of network status and resource utilization. In the exclusive execution, the DNW is the only application that gains access to all resources given by PCloud, while the mixed execution allows other applications to simultaneously access such resources. We use OpenCV 2.4.2 with the Eigen-face algorithm [72] to build this application.

Evaluation Scenario: When some user monitors a street closed to his house, he

Table 1: The number of recognized faces per each participant in the DNW

Name	Faces in Facebook	Standalone phone
A	32	10
B	23	16
C	24	9
B	35	11
B	14	5

runs the DNW application (app) on his mobile device. The app captures people's image on the street periodically (every 3 seconds) and figures out who they are. These results in a capacity query sent to each configuration of the face recognition service: (i) located on the device, (ii) a set of user-owned devices, and (iii) remote cloud. For this use case, our current heuristic for resource allocation is initially to choose the nearby resources for services unless the remote one, i.e., the Amazon instance, offers a performance index that is twice as high. If services on nearby resources is not available, the heuristic is to attempt to work with remote cloud services. Compute power on a mobile device is used only as a last resort.

Results of The Scope of Data: Five fellow students helped run the experiment by installing the required Facebook app for the data aggregation service, resulting in the remote store containing all photos and tags on their respective Facebook accounts for identified friends, the idea being to emulate several parties living in the same neighborhood. We assume that a user A, in Table 1, has a mobile phone participating PCloud. The numbers of faces listed in Table 1 use photos from the user data aggregation service attached to all participants (i.e., A, B, C, D, and E) since all of them are in the *DeviceShare* group in Facebook. The numbers mentioned in Table 1 refer only to those that are sufficiently clear to permit our face recognition software to correctly identify faces. Note that the total number of distinct faces accumulated by the data aggregation service is 105 after removing duplicated results, thereby demonstrating its ability to provide a wider scope with a broader set of data set than that available on any compared to any specific device.

Results of Performance: We first evaluate the elapsed time needed to identify a person's face via the neighborhood watch app. As mentioned earlier, the neighborhood-watch app sends a face recognition service request every 3 seconds for 5 minutes in the mixed execution environment. Then the PCloud runtime and Cirrostratus build a PCloud instance and allocate a resource for the service either on one of nearby devices (`N1` or `N2`) or on the cloud (`EC2`). Figure 7 is a snapshot of the elapsed time to process one request. As seen in Figure 7, the service located on the nearby resource, Config. (ii) (`Nearby`), shows the best response time when each service can identify a given face. With Config (ii), transmitting states and user data consumes 331ms out of the total elapsed time of 1335ms (about 24.7 % of time). This includes time where the PCloud runtime constructs a PCloud instance to meet the requirement of the Face recognition service before transmitting them. However, most of which time is influenced by the network connection status. The time is less for nearby devices, compared to using the remote cloud due to the network delay.

Figure 8 shows that power consumption is substantially higher when the DNW runs on that single device. This is in stark contrast to results shown for the `Nearby` and `Cloud` configurations: the energy consumption on the mobile device of Config. (ii) and (iii) (`Cloud`) compared to (i) (`Standalone`) is reduced by 74 % and 77 %, respectively. Even if we omit CPU usage on the mobile device for each configuration, it still follows the power consumption plot in Figure 8: this means that power savings are due to the fact that compute-intensive face recognition functions can be successfully offloaded to a suitable computing resource.

During the experiment, resource allocation takes place 19 times, due to changes in network condition and resource availability caused by other workloads being run. Figure 9 shows the user-experienced total response time taken to obtain recognition results. If the resource is located on the cloud, network delay is three times larger than when using the nearby resource, as seen in Figure 10, but the increased computational

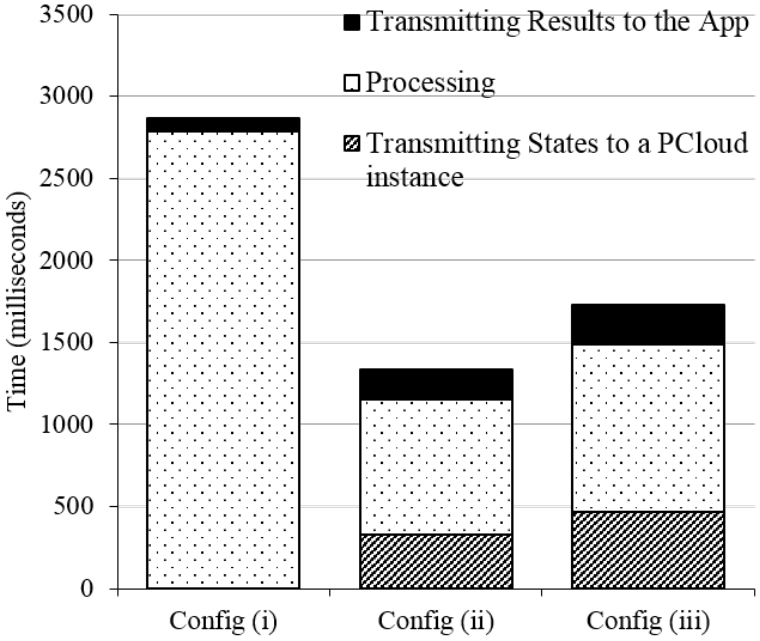


Figure 7: User-experienced response times

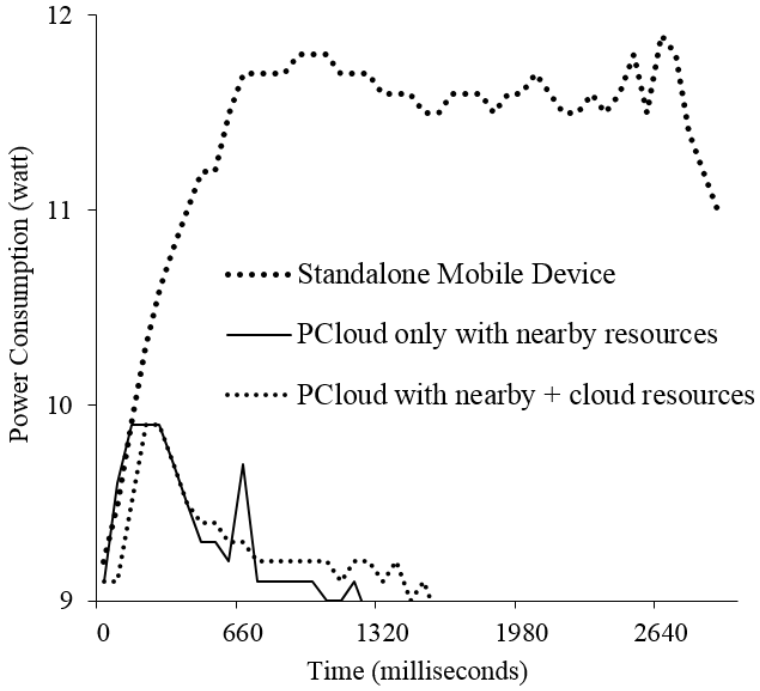


Figure 8: Comparison of power consumption

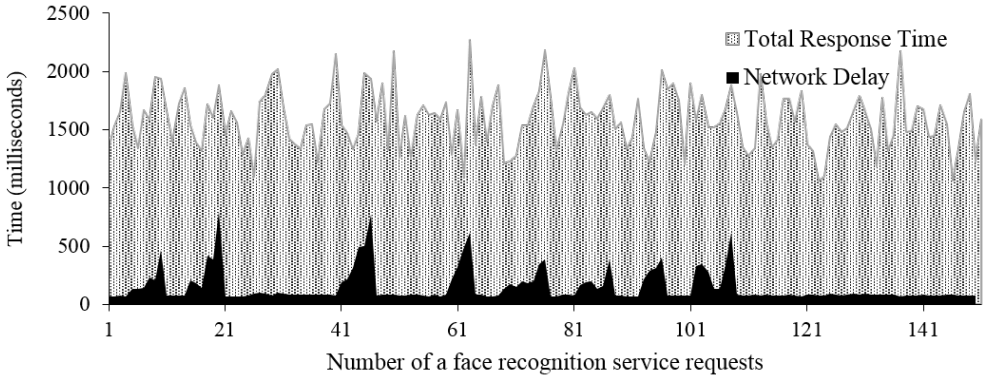


Figure 9: User-experienced total response times for face recognition requests

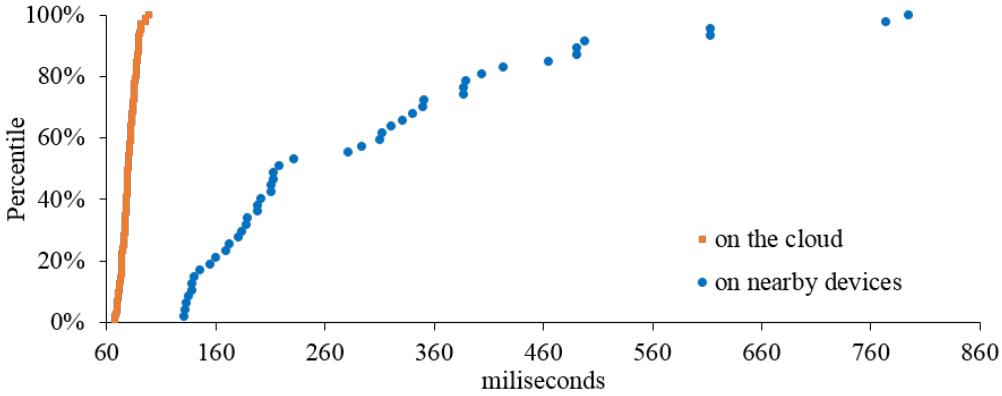


Figure 10: Cumulative distribution functions of network delay

capabilities of the cloud successfully offset that additional delay. Comprehensive comparisons in Figure 11 show that for this application, user-experienced response times are almost identical no matter what resource is given to the service.

3.7.2 Display Sharing

Test-bed Configuration: We evaluate the costs of projecting a small screen on a smartphone to a large nearby display. We use a 23-inch monitor with full HD resolution (1900 x 1080) attached to a desktop PC, participating in PCloud interacted with the master in Cirrostratus, all connected to the mobile device via 802.11n. The Atom platform and Nexus 7 offer 1024 x 600 and 1280 x 800 resolution, respectively. The authentication service runs on an Amazon `t1.micro` instance hosting a Facebook app.

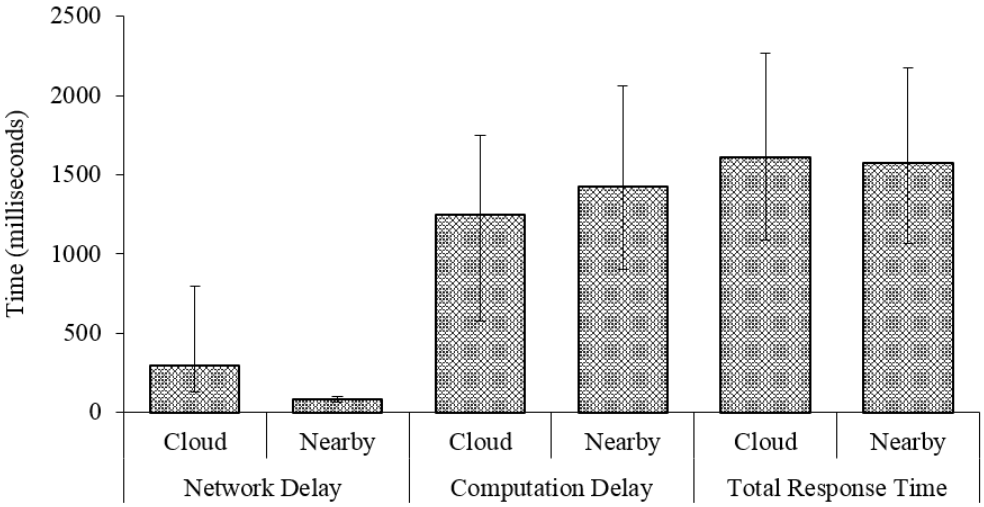


Figure 11: User-exprienced average delays

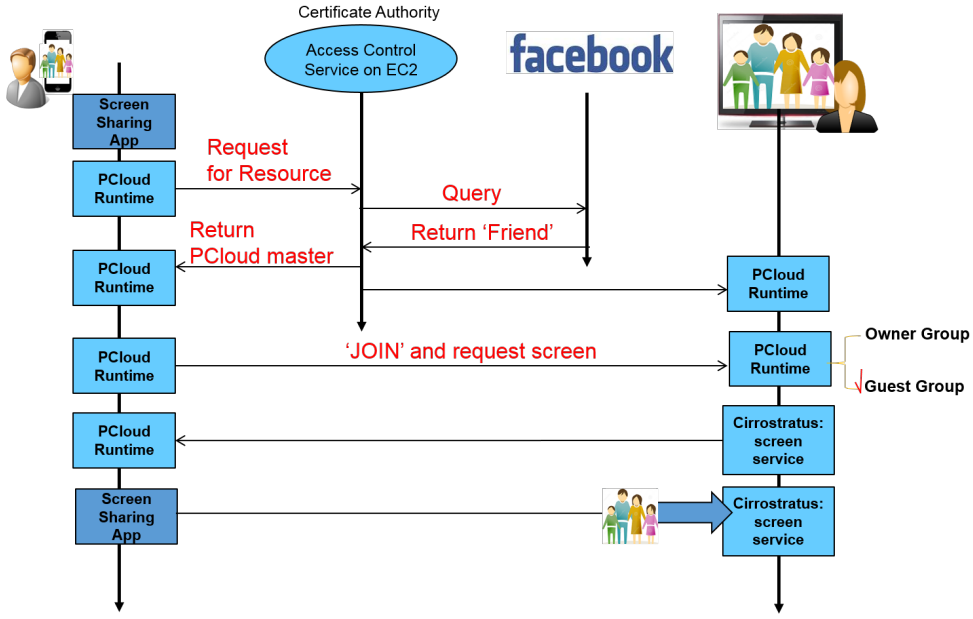


Figure 12: Evaluation scenario procedure

Evaluation Scenario: Suppose that Alice shows video clips of her newborn baby when she visits Bob's house. With a PCloud instance, if a PCloud environment is set up at his home, she can easily depict this clip, as long as both Alice and Bob are friends on Facebook. In this case, Alice asks Bob to share the large TV with her, and he places her into the *Device Share* group in his Facebook account. Assuming she already installed the Facebook app for the authentication service, when she starts the app, Facebook leads her to a server that hosts the app. Now the app shows a list of people who are willing to share their device's capabilities by putting her on their *Device Share* group. She selects Bob on the list. If this is her first use of the authentication app, the authentication service generates her certificate based on PKI (X. 509 specification). The service also provides the IP address of Bob's Cirrostratus master along with her private/public key pair. Authentication is completed when the PCloud software in Alice's phone starts a connection with Bob's Cirrostratus master, with all subsequent communications transmitted via a secure M-Channel connection. Using this connection, Cirrostratus is allowed to share with Alice's device a list of all permitted capabilities for guest users. Our current implementation only supports two groups – the owner and guest group, where only a member of the owner group can adjust the capabilities being shared. The Personal Cloud software (`agent`) in her phone, then, chooses the desired display, lets the capability service know about its selection, and finally, starts transmitting the mobile phone's screen to an appropriate PCloud instance. Before the transmission takes place, the PCloud runtime in Bob's house initiates a PCloud instance including the large TV and begin with the remote display service on the instance. Then the desktop machine starts its VNC [12] receiver as the remote display service to project Alice's screen on its display. Figure 12 depicts each step in PCloud for the display sharing use case.

Evaluation Results: Table 2 shows the elapsed time from when Alice sends a request for display sharing for the first time to Bob's Cirrostratus. In this case, Alice

Table 2: Elapsed time in milliseconds for display sharing

Task (message)	From	To	Elapsed Time	Std. Dev.
Initiate a certificate	MO	S	88.2	27.6
Return the certificate and key pairs	S	MO	213	35.3
Authentication	All	All	405	41
Send a display sharing request	MO	SM	140	66.3
Return a list of available capabilities	SM	MO	293	117.3
Notify a selection of a display	MO	SM	179	55.8
Initiate a VNC connection	SM	MO	153.3	80
Total Elapsed Time			1471.5 ms	

MO : a mobile device
S : the authentication service
M : a Cirrostratus master
All : all devices joining a device cloud

has to first create her certificate. As seen in Table 2, it takes 1471.5ms to be ready to project the screen on her mobile phone to Bob's display, likely superior to a case in which Alice and Bob try to wire these devices manually. Further, with PKI and SSL connections obtained by the authentication service, such sharing is done in ways that respect access rights and provide end user security.

3.7.3 Discussion

Interesting to note about the neighborhood watch application is that while local resources offer low latency response, the increased data scope of remote resources can offer improved accuracy, and in addition, for this application, the substantially faster remote resources can also hide some of the additional network delay seen for requests to the remote cloud vs. local machines. Such trade-offs are a general attribute of the PCloud approach, encouraging future work combining performance indices capturing such trade-offs, with networking data, and with indications of what may be most important to end users when services are run, i.e., user intent. In fact, it is because of the flexibility offered by PClouds that such trade-offs become possible, permitting mobile devices to opt for *best-fit* resources in their current environments. The PCloud

approach supports such actions via continuous system monitoring to understand the capabilities offered by currently reachable resources (the performance index in this case) and network connectivity.

Display sharing via PClouds demonstrates seamless and secure ways to leverage nearby resources, supported by SNS-based authentication and access control. Offline certificates [34] can be used when remote SNS services cannot be reached. More generally, efficient operation disconnected from remote clouds is an obvious advantage provided by PClouds, for which we are currently considering additional Cirrostratus enhancements that can distinguish important from less important services when a user wishes to run multiple services on potentially scarce nearby resources.

Personal Clouds are shown capable of substantially augmenting the capabilities of mobile devices, and they can alleviate their limitations, including lack of performance, limited battery lives, and constrained form factors. They can also deal with the restricted the *scope* of the data currently resident on each device. In contrast to remote cloud services used by mobile devices, PClouds can also augment device capabilities through the use of nearby devices. The outcome is not only increased storage and computational capacities, but also the creation of entirely new functionalities not available from remote services, such as the ability to present on large displays, the potential to share content not resident in remote clouds, and others.

PClouds leverage Social Network Services (SNS), i.e., those provided by Facebook, for authentication, access control, and for secure device interaction. Their hypervisor-level realization includes fully virtualized devices like home desktops or server systems as well as non-virtualized mobile devices like the Android tablet used in this paper. Performance evaluations show PClouds capable of augmenting device capabilities to improve their performance as well as enhance the functionality seen by end users.

In summary, PCloud realizes new design principles discussed in Chapter 2 to extend the idea of VPs. In contrast to the Stratus infrastructure, PCloud can

support non-virtualized client devices like Android smartphones as well as resources from remote clouds and owned by different users. To enrich a resource pool for a VP, resource participation in PCloud is guided by permissions and policies controlled via social network services (SNS), thus making it possible to share resources owned by different users in a secure and safe way. In addition to enriched sharing, PCloud improves applications' scope through its aggregation storage service that takes advantage of the SNS, which allows applications' scope to go beyond the data currently resident on a single device. Finally, PCloud directly accepts an application's request for resources via its *Intent* API rather than uses the policies that STRATUS introduces. In doing so, it enables VPs to more accurately understand and respond to applications' ever-changing resource requirements in a timely manner.

3.8 Related Work

ThinkAir [73] proposes dynamic resource allocation on the server side to maximize parallelism for offloaded workloads. MAPCloud [97] models mobile application as a location-time workflow (LTWs) and maps LTWs onto online resources with 2-tier clouds. Our work also considers two layers of resources, nearby and remote clouds, but PClouds can share fine-grain device capabilities among participants as well as perform compute offloading. HomeOS [51] pursues goals similar to those of PCloud. Like PCloud, it permits users to formulate and enforce desired policies across all HomeOS devices, but it does not exploit SNS to enable the access control and consequent rich methods for device sharing present in PCloud. SBone [109] uses social networks to create an overlay over existing social network graphs for enabling devices to share content, state, and computational resources. Cirrostratus shows the utility of leveraging efficient local area networks, and it can compose a new abstract at the finer granularity of individual device *capabilities*. We leverage well-known authentication technology, with the innovative approach of using the Facebook SNS

as an authentication service. Our approach implements a framework similar to [99], which suggests using Google's OpenSocial and Facebook's Connect to provide APIs for web-based social network applications, allowing users to carry their identities across applications and devices. Using remote resources to augment the compute performance of mobile devices has been studied since [55]. Recent research like MAUI, CloneCloud, Mars [44], and Cuckoo [71] address the question of how to find the compute intensive portions of an application and move those between battery-operated devices and high-performance backend servers. The focus is on remote cloud servers, however, so these systems do not have functionality that describes and can exploit the varied capabilities present in the many and often heterogeneous devices present in the nearby cyber-environments targeted by our work. Similar to our own previous work, Cloudlets [64] use small servers as nearby computing resources to augment mobile devices, via virtual machine migration, but the system does not support non-virtualized devices. [117] suggests use cases similar to ours, arguing that the use of smartphones heavily depends on context, in particular, on other devices and places, and on the situations users are experiencing.

CHAPTER IV

STENCIL: ACCESS CONTROL SERVICE FOR VIRTUAL PLATFORMS

The more devices join a resource pool, the better virtual platform will be created because of the pool's increased set of resources. Consequently, edge cloud infrastructures like PCloud are able to construct better virtual platforms that match application needs than in our earlier work on Stratus . Toward this end, PCloud offers an SNS-based mechanism so that users can easily share their resources with others. The SNS service assumes such users share a special group on Facebook, as discussed in 3.6.3. Use of that service enhances PCloud 's resource coverage, but also imposes additional burden regarding access policies of each resource on end users. This chapter proposes a novel approach, Stencil, to ease the difficulty and complexity when users handle access policies for all their resources on edge cloud infrastructures.

4.1 Goal

With high diversity existing in today's computing devices as well as the enormous number of sensors with which users can interact, it becomes more challenging for users to properly decide on the different access privileges granted to individual resources when sharing them with others. Hence, the goal of Stencil is to help such resource owners easily set up access privileges for their own resources, while ensuring safe and secure operation.

4.2 Approach

In Stencil, fine-grained access control is realized by a reference monitor that enforces access policies to any resource on a VP to be granted only to parties authorized to

do so. It also provides resource owners with a runtime recommendation for access privileges, determined from (i) social relationships between the invoker of a resource and its owner and (ii) the current context in which a request is made.

4.2.1 Authorization and Recommendations

A call to a resource in a VP by user's application (i.e., principal) must not succeed if that principal does not have suitable access permissions to the physical resource in the VP. For every call, therefore, a VP should check the caller's access permissions and provide the resource only if access is permitted. To do so, STENCIL uses discretionary access control (DAC) [96], implemented via cryptographically protected capabilities for all resources for virtual platforms. The access rules, i.e., policies, realized in this fashion are formulated by resource owners to control who (i.e., some principal) is authorized to access certain operations associated with the resources in question. There are explicit operations for creating policies, granting and revoking access rights, and to restrict delegation.

While fine-grain protection is important, it is difficult to formulate and express such protection policies in environments targeted by STENCIL [101]. We address this issue by providing to resource owners an online recommendation service for access permissions, leveraging the wealth of information about potential principals available on social networks [58] and data about the current context in which the application is operating [27].

4.2.2 Secure Offloading for Applications

Applications may require a wide array of software services that process their tasks. If they run on the battery-operated devices, some edge cloud infrastructures like PCLOUD and Cloudlet allow such tasks to be offloaded to richer resources for the sake of performance and battery life. Such tasks are run either on the device on which the app is running or on any accessible ambient computing resources. Since

they are preferred to run in an isolated environment for its safe execution [48], where the resources allocated for those services and sandboxes are controlled by a mechanism cooperating with the underlying edge clouds like PCloud and Stratus [67, 68].

4.3 Architectural Design

Stencil has two main components for its design to realize the aforementioned technical approaches, the reference monitor and recommendation service. The reference monitor enforces an access policy for each invoked resource while the recommendation service, which is backed by information derived from the social network services (SNS) and the current context, provides end users with an easy way to construct such policies. It also performs mutual authentication.

4.3.1 Reference Monitor: Access to Resources on Edge Clouds

Internal runtime representation of the access controls mentioned above is a list of access rights to services, resources and data, which is enforced by Stencil's internal **reference monitor**. If, for example, the user drives up to her house, the reference monitor would provide full access to all in-house edge cloud resources, e.g., her home PC, home sensors, etc. If the user drives up to her friend's house, such access is limited to only those resources to which her friend has granted her access, e.g., the friend's large screen display for jointly viewing vacation pictures. The FaceRecognition service in 3.6.3 is another resource made available in this fashion, where within say, a home edge cloud, the home owner has access to and can run a rich set of services (and the resources on which they run), e.g., to determine whether a person visible on a home security camera is someone known to the homeowner or an unwanted intruder.

4.3.1.1 Authentication

Access controls begin with mutual authentication activities between two principals, i.e., a user running an app and the owner of resources that the app tries to access.

In Stencil , those activities involve a Facebook-based app installed by the user on her Facebook account and a server hosting this app on an Amazon EC2 instance for a trusted key server.

4.3.2 Recommendation Service

To assist resource owners create access policies, Stencil's recommendation service provides them with a preset access policy based on information acquired from their SNS and the current context in which a request is made. Stencil assumes that the owners are willing to share their resources with those who are closed to them in the real world. In Sociology, such people are referred to as having a strong social tie with the owners. This tie, however, is hard to measure in the real world to construct access policies. Hence, Stencil's recommendation service uses recent studies proposing certain models for predicting such social ties from the interactions observed in an SNS like facebook. For the relationships that are unable to be captured by the SNS, the recommendation service refers to the context in which a request is made. The SNS information helps the service predict the real strength of social ties, and the context information captures the situation that goes beyond social ties, which means that their complementary nature can lead to a more accurate recommendation. The owners can accept or customize this recommendation to make their own polices, or they can simply ignore it.

4.3.2.1 Social Network Service

Recent studies [58, 59, 60] present models that predict actual social relationships, **social ties**, from the interactions between participants observed in social network services (SNS). Such models derive predictive variables from SNS and then use them to estimate the strength of social ties in the real world. Stencil adopts this approach by (i) periodically inspecting a resource owner's SNS interactions and then (ii) using these observations to predict the owner's social ties to other individuals with which

he/she interacts. This prediction, then, is the basis for constructing a set of recommendations for access permissions to the owner's resources. The owner can use these recommendations for making final decisions about granting access permissions.

4.3.2.2 Context Information

An SNS can capture many, but not all social relationships relevant to access to resources. Additional information of value for deciding on access permissions include the context in which access requests are made and the intent behind making those requests [36]. A resource owner may be visiting a gym, for instance, wishing a trainer to have temporary access to her/his personal health sensor. In STENCIL , such context information [75] is captured with Stencil-specific data that can include access to the owner's online calender (e.g., gym appointments), SNS events, and physical sensors like the owner's GPS location via a smartphone [27, 35]. A simple example is one in which SNS relations do not provide a visitor to your home with sufficient permanent permissions to access some home device, like a large-screen display, but additional context derived from an appointment noted in a shared calendar may temporarily provide such permissions.

4.4 Implementation Detail

4.4.1 Access Control and Policies

Access controls enforced by Stencil's **reference monitor** are inherent to how the resources in a given VP are used. Such access controls begin with mutual authentication activities between the mobile device user and any other users (e.g., the homeowner) with whom she might want to interact. In our implementation, those activities involve a Facebook-based app installed by the user on her Facebook account and a server hosting this app on an Amazon EC2 instance acting as a Certificate Authority (CA), implemented with an X.509-based public key infrastructure. This operates as follows. The app, once installed, interacts with the CA running on the Amazon EC2

instance, to verify an identity of a edge cloud and a person. The EC2-based CA, then, holds the Facebook user's certificates for future use. The EC2-based CA only manages the certificate for those who install the app. Each individual edge cloud and the guest device will consult this CA to mutually check if the edge cloud is really the one that the guest device wants to join (and vice versa), and this mutual identification should happen before the access control.

The reference monitor determines what access policies (if any) exist for the user, i.e., the invoker of a specific resource. This determination is carried out by Stencil's **recommendation service**, which (i) looks up the social ties shown in Facebook between the user (i.e., the invoker) and the other person involved (e.g., the home-owner),and (ii) checks for additional context information available for that user. An example of such context is an event, noted in the user's event calendar, where the user is a scheduled participant in a shared meeting with the owner.

The resulting access recommendation, i.e., the access policy to be applied, then, is based on social tie (e.g., how well do I know the owner?) and on context (e.g., are we both attending the same scheduled event?). The policy determined in this fashion is enforced with every access by the invoker to the resources of an edge cloud. The outcome is fine-grained access control in which different access policies are enforced for every invoker. Policy enforcement is efficient, as the reference monitor issues an access token to the invoker based on a given policy, and then, every resource request uses that access token when interacting with access control (which checks the token). An additional optimization implemented in the current system skips such explicit checks for new requests made by a user for the same resource within 1 minute of previous requests.

The current implementation of the recommendation service uses 10 of top 15 social tie prediction variables defined by [58], but we can easily append others to the current service. To create reference policies from those variables, we cluster the

owner's friends on Facebook into different groups using the Jenks algorithm [69]. Each group is mapped to a different reference policy, and these groupings (and reference policies) are presented to the owner as recommendations. The owner can accept them as is, or he/she can customize them as desired. Context is managed similarly: the recommendation service again defines a suitable reference policy and makes it available for inspection and possible modification by the owner. To capture where a request is made, the current Stencil on mobile devices should report its location from a GPS sensor on devices when it requests to connect an edge cloud belonging to others. Beyond using location data, current applications with Stencil use context determined by event pages on Facebook and the Google calendar service. An access token resulting from this context information, which is called a guest policy, is required to renew every 2 hours. Further, while such context can be checked rapidly, estimation of social ties from prediction variables is slow, in part because it must walk through and collect all social traces on the owner's Facebook account (to understand the owner's ties to other users). As a result, the recommendation service only periodically updates its social tie estimates, according to settings controlled by the owner, but captures context information immediately and on demand.

4.4.2 How It Works

Suppose that three persons, (e.g., Alice, Bob, and Charlie) have an appointment at Alice's house (now Alice is the owner of an edge cloud) and Alice is a friend with Bob on Facebook, but not with Charlie. Since the Stencil periodically evaluates the social relationship between Alice and Bob on Facebook and gives Alice a recommendation regarding to Bob's access to her edge cloud based on such evaluations, Bob's application may now access Alice's resources as long as both are so close on Facebook. The current implementation clusters all friends on Facebook into four different group, and provides recommendation based on the cluster that the each friend belongs

to. However, Alice can still modify the policy for B manually. For any requests from Charlie's applications, STENCIL first checks the cluster that the user, Charlie, belongs to and ends up finding Charlie does not have any access right for Alice's resources unless she previously sets it manually. Since STENCIL can recognize the location of Charlie , it also looks at Alice's calender and Facebook event pages to check if Alice and Charlie have any appointment where Alice's resources are located (i.e., her house in this case). It will see her appointment with Charlie at her house, then recommend her to allow Charlie's applications to give a pre-defined guest policy. Then Charlie's app will be able to access the resources residing in Alice's edge cloud. In addition, Bob's applications will also go through the same process with Charlie's applications if Bob will fall into the cluster than is allowed to access to certain resources.

4.5 Motivating Use Cases

A use case for STENCIL is a digital neighborhood watch application (DNW) [42]. Such applications must access sensors owned by many parties, thus requiring authorization for such accesses, and they must implement potentially complex privacy-conserving algorithms for the homes' cameras [38], smoke detectors, and environmental sensors being used. The DNW application with STENCIL show dynamic authorization and sensor as well as other resources protection functionality, and it uses *edge cloud resources* to run the potentially complex services associated with such accesses, including the resource-access mediation methods themselves.

4.6 Experimental Evaluations

We extend the DNW application 3.3.1 to work with STENCIL for our evaluations as follows. The application uses the same test as described in Section 3.3.1. The only difference from the PCLOUD case is to add STENCIL to authorize app's access to Alice's camera. So we can identify Stencil's overhead.

4.6.1 Test Settings

A Facebook account used for the evaluation has 2675 postings with 3458 comments and 2270 likes. The DNW app runs on a workstation with Intel i7 950 CPU, 8GB memory, and AMD Radeon 270X GPU. Alice's camera node is realized by using TI OMAP5432 CPU and 1080P USB camera. Finally, for the authentication purpose, our trust key server is deployed at an Amazon EC2 m3.medium instance.

4.6.2 Recommendation Service

Table 3 shows how quickly a recommendation is offered by the recommendation service. The Facebook-based recommendation responds very quickly because STENCIL caches the estimation results from the model at a local PCLOUD , which means the recommendation service in STENCIL does not evaluate the model every time a request comes because its execution time is too long to use it in realtime. With the test settings, it takes 453 minutes since evaluating every single social interactions on Facebook to accurately generate these recommendations is a demanding task, its weekly recomputed results are cached and reused, with the assumption that social ties are unlikely to change over that time period.

4.6.3 Digital Neighborhood Watching

The digital neighborhood watch app requires controlled access to distributed sensors. When deployed at the DNW server (Table 8) owned by Bob, for example, it attempts to access web cameras belonging to Alice. Upon such an attempt, SOUL access control managing Alice's resources recommends to her whether the access should be granted or not, based on the social ties between Bob and Alice. Figure 13 shows the strengths of those ties and the clustering of the groups for recommending access policies based on ties, using data is extracted from Alice's and Bob's active Facebook accounts. In this case, Alice has 121 friends on Facebook, and the access control service sorts them into three groups based on the the strength of the tie. If Bob falls

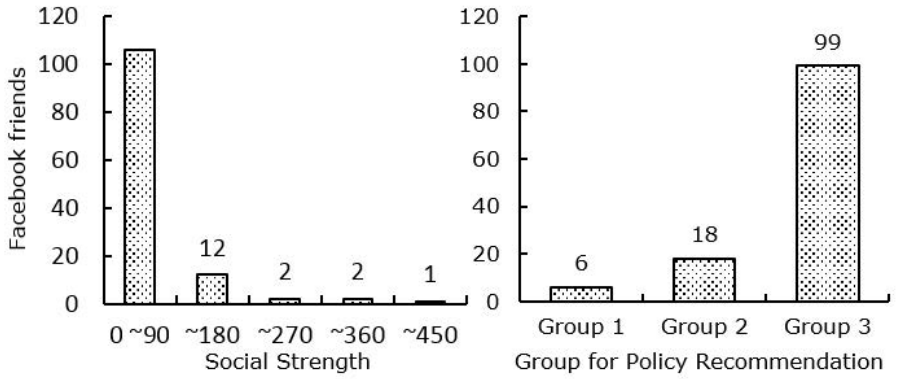


Figure 13: Evaluation of social tie and the recommendation service.

Table 3: The elapsed time to create a recommendation

Recommendation	Time in milliseconds	Std.Dev
Facebook-based	6	0.3
Context-based		
– Via Facebook Event pages	123	56
– Via Google Calendar	90	24

into a proper group, the recommendation service recommends to Alice that she allow Bob's access to her web cameras, which Alice may accept or reject.

Finally, once access to any participating camera is granted, the app tries to identify any person seen in the picture from the cameras. Table 4 shows the entire process to take 1.8 seconds, i.e., to identify faces after the camera captures a picture, which shows 6.7% additional processing time for the authorization conducted by Stencil .

4.6.4 Discussion

The applications interacting with the edge clouds that provide transient resources visible only in certain locations or environments must dynamically acquire the rights to access them. Stencil intends to provide such applications with required runtime permissions, including to resources extracted from nearby ambient resources and/or the remote cloud. It also minimizes user's manual input to create access policies for individual resources via its recommendation service leveraging the SNS and capturing

Table 4: Results for the digital neighborhood watch in milliseconds

Tasks	STENCIL Time	PCLOUD Time
Mutual authentication	534.8	534.8
Camera-access invocation		
– Open a camera	111.5	104.3
– Read a captured image	160.3	152.9
– Close the camera	19.7	15.8
Face identification		
– Detection	673.8	673.8
– Recognition	300.1	300.1
Total elapsed time	1800.2	1781.7

the current context. Our DNW application shows that STENCIL efficiently supports resource sharing between different ownerships of the resources.

CHAPTER V

SOUL: MOBILE APPLICATIONS IN A SENSOR-RICH ENVIRONMENT

Interacting with sensors presents numerous challenges, particularly for applications running on resource-constrained devices like smartphones or tablets. The Soul abstractions in this chapter addresses the challenges faced by such applications through virtual platforms. Toward this end, Soul expands the virtual platform realized by PCloud for enabling novel sensor-related mobile applications. Soul creates a new sensor abstraction on top of virtual platforms, which allows such applications to easily (i) access device built-in and environmental sensors with the level of convenience needed for their ubiquitous, dynamic use, and only by parties authorized to do so, and (ii) scale their sensor use, given today's multitude of sensors.

5.1 Goal

Mobile devices are now the medium of choice for people to interact with each other and their environments, whether carrying out business or personal tasks. Such interactions can be enriched by the ever increasing availability of on-device and ambient sensors, including to for customizing user interactions to current contexts and needs. Recent high-end smartphones, for instance, have more than 10 embedded sensors, and there are already 6 sensors on average in roughly 30 to 40% of today's mobile phones [14, 24], and a similar trend is seen for emergent wearable devices. Mirroring the growth in device-level sensors, there is also an increasing presence of sensors in users' external environments, ranging from those in controlled settings like homes, cars, or hospitals, to those in public settings like sports venues, entertainment, or parks [11]. Tesla's

Model S, for example, has 12 sensors, and Google's driverless car is known to use at least 6 sensors just for its obstacle detection unit. Smart homes can have 1000s of sensors for providing home security, automation, and entertainment [11].

SOUL addresses the challenges faced by mobile applications (apps) that seek to leverage and use the dynamic sets of sensors present on mobile devices and in the environments where they operate. Three particular issues faced by such apps are (i) the diverse nature of sensors, reflected in the need to use per-sensor protocols for interacting with them; (ii) the computational and data management challenges in interacting with sensors, particularly for applications running on resource-constrained end devices like smartphones; and (iii) the dynamic nature of sensor presence, as users move in and out of their proximity and run applications requiring their dynamic access.

Given the issues identified above and the resulting complexity in writing sensor-driven apps, it is not surprising that most of today's apps do not truly leverage the rich, evolving on-device and remote ambient sensor ecosystem. In fact, less than 0.5% of the apps available in the Google Play Store in 2012 actively used sensors [24], and this situation continues to persist. An analysis presented in this paper shows that sensors are used in less than 2% of those millions of apps!

The goal of the Soul abstractions is to make available to applications the sensors present in their current surroundings, with application services running on systems that range from individual end user devices, to nearby computing resources (e.g., a home PC), and/or the remote cloud as Figure 14. Another goal is that in such settings, Soul not only protects dynamic sensor access and use, but also offers recommendations to sensor owners about suitable access permissions for the principals involved, based on the owner's social network interactions with those principals and the contexts in which requests are made by improving Stencil for sensors and associated resources.

5.2 Contribution

Addressing the paucity of sensor-rich applications, the SOUL (Sensors of Ubiquitous Life) and its abstractions make it easier for such apps running on resource-constrained devices to interact with and manage the dynamic sets of currently accessible sensors. SOUL

- **externalizes sensor & actuator interactions and processing** from the resource-constrained device to nearby and remote cloud resources, to leverage their computational and storage abilities for running the complex sensor processing functionality;
- **automates reconfiguration of these interactions** when better-matched sensors and actuators become physically available;
- **supports existing sensor-based applications** allowing them to use SOUL's capabilities without requiring modifications to their code; and
- **authorizes sensor access at runtime** to gain protected and dynamic access for applications to sensors controlled by certain end users.

The functionalities listed above are obtained via the *SOUL aggregate* abstraction, which for mobile applications, is their single point of access to sensors, actuators, and associated software services. This abstraction is realized by middleware that leverages *edge cloud* infrastructure – in our case, the representative PCLOUD [67] system – to efficiently run SOUL aggregate functionality. The outcome is that with SOUL, computationally or storage-intensive sensor-related data management and processing tasks can be externalized from the smartphones that invoke SOUL, offloading these tasks to run *anywhere* in the edge cloud, including on currently available edge cloud machines and in the remote cloud. For such actions, sensor and resource accesses are guided by dynamic access permissions.

Key to SOUL's *aggregate* abstraction is the insight that sensor access can be virtualized, where instead of directly communicating with some physical sensor, the sensor

interacts with an online datastore, and apps interact with the virtualized sensors presented to them by that store. The store interacts with the diverse nature and dynamic presence of current physical sensors and virtulizes them such that Android provides apps with sensors so that SOUL can supports the existing applications without requiring them to be reprogrammed. While SOUL transparently supports existing apps, apps using the SOUL API can fully utilize SOUL's features. For example, SOUL's automated reconfiguration actions can shield apps from the need to understand what physical sensors are currently accessible. Further, the virtual sensors exposed by the store need not map one-to-one to specific physical sensors, and the processing needed to create useful virtual sensors need not be done by the apps running on resource- and energy-constrained smartphones. Instead, edge cloud resources can be leveraged to carry out these computationally costly processing tasks, and the datastore can run the potentially costly protocols required for sensor interaction. Finally, since all of this functionality is implemented in ways that maintain, for existing apps, the programming model in the current Android Sensor Framework, the existing Android applications using sensors continue to run without the need for modification.

With SOUL, a single smartphone app can interact with 100s of physical sensors, actuators, and associated services, minimizing the impact on a device's battery life and performance. The evaluations in this paper show, for instance, that apps utilizing SOUL see reductions of up to 95.4% in access latencies for on-device sensors, compared with direct accesses to such sensors on the actual Android smartphone, and they can scale up to using 100s sensors without notable increases in smartphone power consumption.

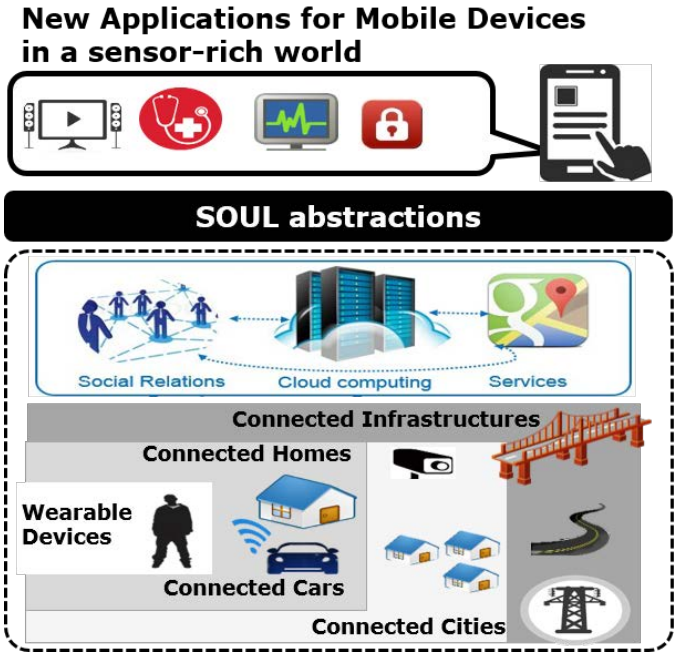


Figure 14: The goal of SOUL

5.3 Sensor Use in Mobile Apps

With the advent of a world of a trillion sensors and mobile devices, a new class of applications has been predicted to emerge, providing end users with personalized services based on the precise capture of their current contexts and intents. Those apps, however, must actively interact with the numerous sensors present on mobile devices and in their current environments, e.g., to extract user's personal contexts or environmental conditions. Unfortunately, today's reality is that most mobile applications (apps) use only a few physical sensors, despite the fact that the devices hosting such apps are themselves sensor-rich. Estimates [14, 24] are that apps using at least one sensor in Android devices with typical devices equipped with six sensors on average are just 0.5% of all available apps in 2012. It is a surprisingly low rate considering the nature of such devices, which is capable of capturing current user context with their multiple sensors and feeding such information to the apps. This section describes more precisely the current status of how apps interact with sensors, enhanced with our own comprehensive study of current apps' sensor use.

5.3.1 Background

Android provides apps with two sensor-related services: (1) the Android sensor framework, and (2) a location service. The sensor framework contained in the `android.hardware` package is a principal means for apps to access raw data from built-in sensors. The framework distinguishes hardware- from software-based sensors, the former directly mapped to specific physical sensors embedded in a device, and the latter emulating desired sensors by synthesizing raw data from multiple hardware-based sensors. An example of a software-based sensor is an orientation sensor combining the hardware-based gravity and magnetic sensors to identify the angle of the device relative to the ground and direction of movement. Sensor availability and access methods vary, depending upon manufacturers (e.g., Samsung vs. HTC), device models from the same manufactures (e.g., Samsung Galaxy S5 vs. S6), and even on Android versions (Galaxy S3 with Android ICS vs. Kitkat) installed in the same device. Finally, the sensor framework may expose multiple sensors for a single physical one, e.g., different sensing rates or resolutions.

Android's location service has evolved from the `android.location` platform API to the Google Places API for Android. With the `android.location` API, applications must manually interact with `GpsLocationProvider`, `NetworkLocation Provider`, and even `SensorManager` to decide on a user's current location. For example, if a user is walking into a shopping mall from an outdoor parking lot, an app using `GpsLocationProvider` suddenly loses the user's location because a GPS signal is lost in the mall, so the app needs to change its location provider to `NetworkLocationProvider`, with the hope that the WiFi or cellular connection may be available. If not, the app considers sensors like accelerometers, orientation sensors, etc. to estimate the user's current location from the last GPS-provided location. In other words, with this API, apps must explicitly deal with sensor availability, thus creating levels of complexity in sensor go beyond their innate functionalities.

Complexity like this is one of the motivations for developing SOUL and in fact, for location sensing, Google has already taken a similar step, by developing the Google Places API that removes from apps the need to interact with individual location sensors, by presenting `FusedLocationProvider` as a high-level abstraction. The outcome is that location sensors are the most commonly used sensors in Android apps, far outpacing other sensor usage in the Google Play store. With SOUL, we provide apps with levels of convenience of sensor use like that offered by the Google Places API. We next delve more deeply into the mobile app ecosystem and its current use of on-device and ambient sensors.

5.3.2 Mobile App Analysis

5.3.2.1 Methodology

Using a tool inspired by recent studies [4, 9, 120] that downloads and analyzes apps from the Google Play Store, we analyze how such apps behave on the Android platform, scrutinizing their bytecodes and manifest files. Our initial study used the top 100 apps in each category on the Google Play Store, resulting in a total of 5,000 apps (on August 21, 2014). We then expanded it to almost all free apps in the Store (750K out of the entire 1.2 million apps on November 20, 2014). Outcomes from the study reported next include (i) the number of apps that use at least one sensor, (ii) the count of sensors used per app, and (iii) the most commonly used sensors.

5.3.2.2 Sensor Usage in Current Apps

As shown in Table 5, for the top 100 apps (in each category of Google Playstore), 81 out of 5000 (1.62%) use at least one sensor. This constitutes only a small increase over the previous estimate of 0.5% reported in 2012. Our more extensive survey of 750K apps in Table 5 do not show any notable differences from the top 100 apps, with 1.92% of those apps using at least one sensor. Further, for apps using sensors, most of them only use a single sensor despite the multiple on-device sensors available

Table 5: The number of apps based on sensor use.

Sensor Count	Apps(top 100, 5K)	Apps(750K)
1	64	12233
2	13	1054
3	3	831
4	0	23
5	1	17
6	0	126
7	0	2
8	0	60
Total(%)	81 (1.62%)	14427 (1.92%)

Table 6: The most commonly used sensors.

Sensor Permission	Counts(top100, 5K)	Counts(750K)
accelerometer	66	13192
compass	15	2391
proximity	8	432
gyroscope	6	1211
light	6	353
barometer	1	331
stepdetector	1	70
stepcounter	1	71
magnetic_field	0	20

to apps. In addition, for both cases (top100, 750K), the accelerometer is the most commonly used sensor, followed by the compass refering to Table 6. All permissions of sensors in Table 6 start with *android.hardware.sensor*. One exception is location awareness, implemented by Android's location services providing location data via a well-defined abstraction to apps discussed in Section 5.3.1 without requiring them to directly access sensors in the sensor framework. Their common use (in almost 30% of the top 100 apps) suggests positive outcomes, i.e., increased sensor use, from developing methods like those realized by SOUL.

5.3.2.3 Discussion of Study Outcomes

Evident from the statistics reported above is the fallacy of recent predictions that mobile devices will naturally become hubs in a sensor-rich world. In theory, many apps could benefit from leveraging all sensors on devices and in a user's current environment (e.g., sensors from wearable devices, homes, and cars). In practice, this is not the case, and it remains unclear why today's apps do not actively leverage even the potential utility of the sensors on their own devices. SOUL addresses this issue by seeking to improve mobile apps' interactions with sensors, in ways that reflect recent lessons from the location service. The following case study shows one potential cause of the dearth of sensor usage, the fragmented nature of today's sensor ecosystem in Android.

5.3.3 Case Study: Temperature Sensors

As an example of temperature sensors, Listing 5.1 illustrates the current complexities of sensor use experienced by Android apps. Since they are not one of eight sensors officially defined in the Android SDK, apps using them must manually retrieve a SensorManager object and its information via the `getSystemService(SENSOR_SERVICE)`, first obtaining a list of existing sensors through `getSensorList(TYPE_ALL)` and then checking the types of sensors in the list, to find either `TYPE_TEMPERATURE` or `TYPE_AMBIENT_TEMPERATURE` depending on the Android API level. Additional issues arise [1, 8, 14, 24] from the facts that (i) device manufacturers may define different ways of accessing the same sensor, sometimes even for different generations of the same products, and (ii) app developers seeking to use sensors have to handle different sensor vendors and their diverse products.

Explicit discovery prior to sensor use, coupled with heterogeneity even in simple cases like the temperature sensor outlined above, causes apps that seek backward compatibility to forgo using such sensors. It may also explain why widely used apps

Listing 5.1: An example code of how to check the existence of a temperature sensor

```
1 SensorManager smanager = (SensorManager) getSystemService(SENSOR_SERVICE);
2 Sensor temperatureSensor;
3 for (Sensor sensor : smanager.getSensorList(TYPE_ALL)) {
4     if (sensor.getType() == TYPE_TEMPERATURE ||
5 sensor.getType() == TYPE_AMBIENT_TEMPERATURE)
6   temperatureSensor = sensor;
7 }
8
```

like Facebook do not actively utilize sensors other than the location service to customize content to users' current contexts. SOUL reacts to this issue by improving ease of use for on-device and nearby sensors as follows: (i) tackling fragmentation via a common sensor API (see Section 5.6.1), with backward compatibility, (ii) providing dynamic, protected access to the sensors present in a device's current external environment (see Section 5.4.3.2).

5.4 SOUL Design

SOUL has two basic components: (1) the SOUL Engine residing in the Android platform manages all sensor-related operations, and (2) the SOUL Core, which is a set of modules and middleware interfacing otherwise standalone systems with the cloud. Its current version built for the PCloud edge cloud infrastructure includes SOUL's sensor datastore, access control methods, and resource management.

Applications use SOUL functionality to access sensor data, control sensors, and to run sensor services via the SOUL's *aggregate* abstraction. Specifically, with the aggregate abstraction, applications can create points of access and use for sensors and associated sensor services and actuators, regardless of where these sensors are physically located, i.e., on the app's local platform or accessible remotely. Figure 15 overviews SOUL's design, described in more detail next.

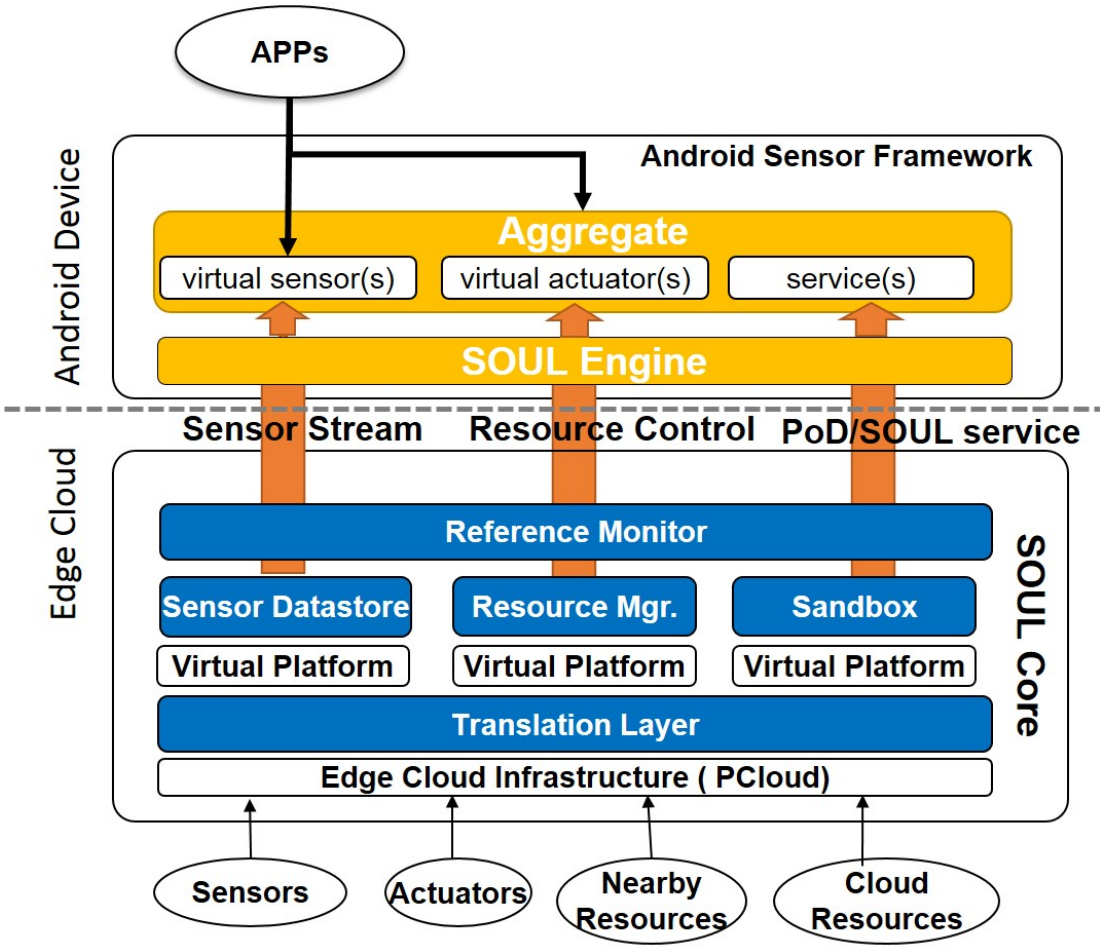


Figure 15: SOUL Design

5.4.1 Design Principles

This section presents design principles for SOUL. The SOUL Engine and Core continue to guide more detailed architecture design and implementation on Android and PCloud .

5.4.1.1 Logical Decoupling

SOUL virtualizes physical sensors, to create easily used sensor abstractions with uniform APIs that replace the custom APIs offered by specific physical sensors. Virtualization also provides flexibility for exactly crafting the abstractions desired by apps. For instance, for sensor processing requiring both current and past sensor data [39],

with SOUL, sensor data can first be placed into a sensor storage service, the *Sensor Datastore*, and SOUL aggregates obtain data from that storage service rather than from physical sensors. Such intermediate sensor data storage permits data regularization and/or time series-based data use [26]. Additional interesting uses of virtualization are to construct entirely new sensor types that fuse or combine [46] several physical sensors into new ones.

5.4.1.2 Flexible Service Execution

Ample previous work demonstrating the need for flexibility in sensor data processing ranges from early results on runtime adaptation for sensor processing [104, 114] to recent work on novel edge cloud functionality [39] and on cloud offloading [85]. Leveraging such results and akin to first storing sensor data in a storage service before delivering it to applications, SOUL offers flexibility in where an aggregate performs its sensor processing: on the sensor itself, on the platform running the app using the sensor, or on cloud processing resources. SOUL middleware obtains such functionality by interacting with edge cloud infrastructures [51, 64, 67], via its *translation layer* shown in Figure 15. We currently use the PCloud [67] representative edge cloud infrastructure, but translation layers for other infrastructures are straightforward to implement.

5.4.1.3 Compatibility with Existing Android Apps

Apps currently using device-level sensors should be able to continue to run, interacting with SOUL without the need to modify their codes. SOUL addresses this issue by exporting to such apps *illusions* of the physical sensors with which they interact. Such illusions are virtual sensors identical to those in the Android Sensor Framework. At the same time, via aggregates, SOUL provides more advanced functionality to SOUL-aware apps. This differentiates SOUL from previous work supporting only new applications written for their new APIs, like GSN [26] and RTDroid [123].

5.4.2 SOUL Engine

SOUL's realization in Android, the SOUL Engine, interacts with apps via SOUL APIs. Invocations of Engine APIs trigger additional important functionality in cooperation with the SOUL Core on PCloud as follows: (i) Access control – the Engine determines each app's degrees of access to desired sensors. (ii) Orchestrated data movement – at the granularity of a group of virtual sensors, it creates *Sensor Streams* that ultimately link sensors to apps. (iii) Externalization – it runs the middleware functions needed to interact with edge clouds, when present. If no edge cloud is available, the Engine runs the app's SOUL aggregates on available local and externally pre-exposed resources on the device hosting the app. Conversely, with edge clouds, SOUL's middleware methods interact with the SOUL residing on nearby edge cloud resources. (iv) Runtime mapping – SOUL permits automated methods to map physical to virtual sensors, with runtime remappings based on changes in user context.

5.4.3 SOUL Core

The SOUL Core is comprised of three modules interfaced with the underlying edge cloud infrastructure, i.e., in our case, the PCloud infrastructure. The first is the Sensor Datastore, which interacts with physical sensors to collect their data and store it into an underlying time-series database. Second, the reference monitor in the SOUL Core enables dynamic permissions to sensor data based on fine-grained access control policies. Lastly, the Core's Resource Manager makes decisions concerning the offloading of sensor management and processing from resource-poor end devices to the cloud. The Core's implementation on PCloud has sandboxing methods, so that it can better control the execution of app-provided, potentially complex and time-consuming sensor processing codes on local or remote resources. Figure 16 also shows the translation layer used to interface SOUL with edge cloud infrastructures

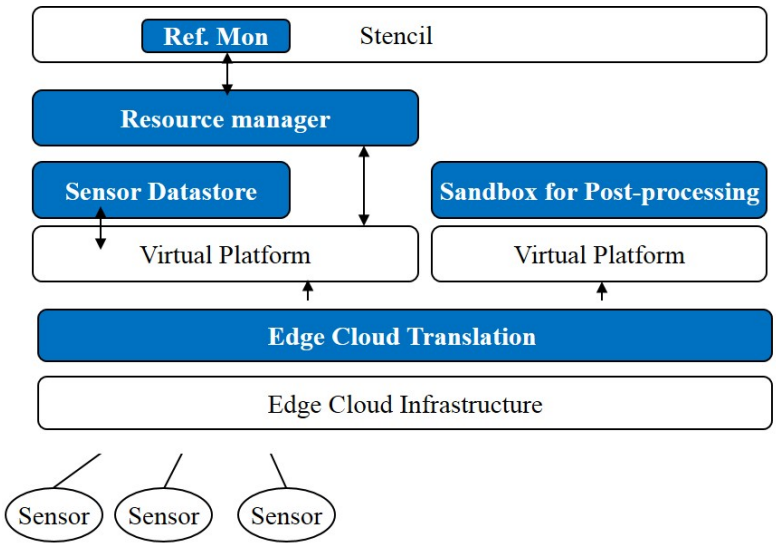


Figure 16: The SOUL Core Design

like PCloud (an alternative implementation for Cloudlets [64] is in progress).

5.4.3.1 Sensor Datastore

As stated earlier, the Sensor Datastore virtualizes sensors, in that sensor data is first placed into the store, then pre-processed to provide apps with different ways to access and use that data. Specifically, upon an app's access to some virtual sensor on a device, the Sensor Datastore constructs a sensor stream to move sensor data between the SOUL Core and the SOUL Engine that ultimately, provides sensor data to apps (in the form of virtual sensors). For example, if an application desires a time-series sensor, the store manager reads the corresponding sensor data and *bundles* time stamps with that data. It can then make available to the app an appropriate virtual sensor (e.g., sensor data along with time stamps). To respond to requests from the apps, the Datastore defines the following operations.

GroupBy. The SOUL Engine API allows apps to combine multiple sensors into a higher level abstraction, to make it easy for apps to scale in terms of the numbers of sensors with which they interact and in terms of the degrees of required sensor processing. Data acquired from such sensor groups is processed by appropriate sensor

service(s) run on the device and/or on edge cloud resources, then placed into the Datastore. Typical properties used in *groupBy* are those based on sensor location, type, etc. The outcome is that apps interact with a single point of control, the sensor group, for any such set of sensors, thus making it easy for an app to see and control them. An example is 'all' light sensors in a single home.

Filter. If apps do not need sensor data to be as precise as that delivered by a physical sensor, they can receive only the data that meets their criteria, expressed e.g., as time windows or sampling rates. The Datastore's *filter* operation implements such functionality.

Bundle. The *bundle* operation can provide additional metadata like time stamps and sensor locations, to enable apps to utilize the sensor data without additional, extraneous sensor or datastore interactions.

Reconfigure. If an initial configuration of a SOUL aggregate must be changed because, say, a user moves to a new location, newly available sensors may be added to the aggregate and others may be removed. The *reconfigure* operation shields apps from such dynamics in the environments in which they operate, resulting in seamless SOUL use across changes in time, location, or even with physical sensor failure.

5.4.3.2 Access Permissions to Operations

With Stencil , Soul enforces access permissions to ambient or nearby sensors. SOUL enhances Stencil to implement dynamic methods for access control for sensors, thus enabling such apps to address the security and privacy issues arising for shared sensing (and sensor processing). Stencil is now integrated with the Soul Core. As already discussed in Section 4, such controls should simply be envisioned as access matrices accessible and checked by the SOUL Engine. How matrix entries are determined, i.e., how it is determined whether or not a certain app should have access to a certain sensor, is a matter of policy. The access control policies realized

for SOUL and the automation actions taken to deal with dynamic sensor arrivals and departures will be described in a future publication, but briefly: (i) virtualized sensors in SOUL aggregates can be designed to export only and precisely the data needed by an app [84, 86, 116]; (ii) sensor owners can define fine-grain permissions for apps to use certain virtualized or physical sensors [101]; and (iii) SOUL assists owners in creating access policies, with automation support that leverages their social network services; and finally, (iv) a sandboxing mechanism is used to safely execute app-provided codes on remote resources that process sensor data [48].

The glance operation: With the emergence of the Internet-of-things(IoT) environment, it is growing that edge cloud infrastructures collect and manage data [63, 65] generated in that environment. STENCIL has a new operation, the *glance* operation, to preserve privacy when apps access sensors. Access controls can be used to permit or disable sensor access, but by controlling which app can use which operations on resources managed by SOUL, it becomes possible to make finer grain decisions that can mitigate the risks to privacy inherent in permitting others to view data from personal sensors. with STENCIL , SOUL handles this via the two distinct operations, *read* vs. *glance*, which export different granularity of data to different invokers. That is, if an app without proper privileges invokes a *read* on a sensor, that access will not be granted, but the app may succeed with its *glance* requests. Specifically, for privacy-sensitive sensor data requiring controls on its exposition to external entities [86], the *glance* operation offered by SOUL aggregates will export only summary data. For example, apps with *glance* see only overall trends rather than detailed sensor data, e.g., private sensor data.

5.4.3.3 Sensor Management and Processing

The other extension of STENCIL is its support for safe execution of sensor-data processing. The descriptions above make clear that SOUL aggregates may require a wide

array of services that implement the rich sensor data processing methods needed by applications. There are two ways to implement such services: (i) by utilizing services already present in the edge cloud [67], or (ii) by dynamically extending a SOUL aggregate's services via app-defined methods for processing the data from sensors in the aggregate (e.g., using JavaScript), described in more detail in Section 5.7.2.1. To enrich whatever sensor processing methods may already be present in an edge cloud, SOUL permits dynamic extension, to allow apps to run their own post processing algorithms on top of PCloud resources, termed Processing on Demand, and when doing so, it sandboxes such codes for safe execution [48], where the resources allocated for those services and sandboxes are controlled by PCloud's underlying mechanism.

5.4.4 Sensor Streams

Apps use virtual sensors much like current Android apps use physical ones, thus preserving the Android sensor programming model. Below that layer implementing the Android sensor programming model, sensor streams bridge between the SOUL Engine on a device and the SOUL Core running on underlying cloud resources. Upon a request from an app, stated for some SOUL aggregate, the Datastore creates a Sensor Stream connecting between the Core and the Engine. It attempts to meet Android-defined constraints on desired sensor data rates and delays, and within those constraints, it also seeks to obtain improved performance by optimizing this stream using sensor data batching. The outcome is that battery-operated mobile devices are shielded from some of the potential overheads of using physical sensors (e.g., battery drain discussed in Section 5.8.2) ; instead, these overheads are shifted to PCloud resources running the Datastore. These optimizations shield apps from potential overheads they might otherwise experience for sensor access, but they do not prevent the datastore from exporting sensor data with the actual precision offered by a physical sensor. This is because all sensor nodes are required to send their raw sensor data to

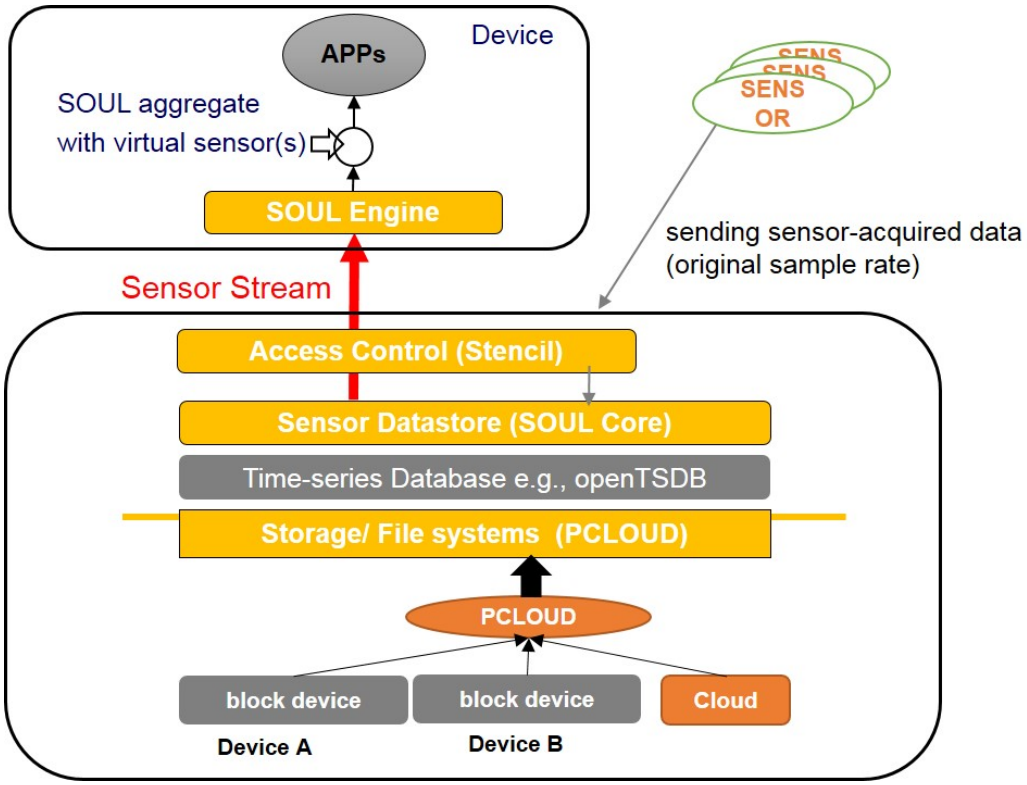


Figure 17: The SOUL Engine and Core connected via Sensor Stream

the sensor datastore at their native sampling rates, via the *writeData* call defined by the PCloud egde cloud. For sensors resident on the Android devices, upon an app's request, the SOUL Engine first transmits on-device sensor data to the Datastore, and then obtains sensor data for apps as for all other sensors. Figure 17 shows how each sensor stream connects the SOUL Core on a device with the corresponding Engine on an edge cloud.

5.5 The SOUL Engine

For sensor-based apps, SOUL aggregates are the single points of use regarding sensor access, actuation, and processing services. Apps access their SOUL aggregates via the *Engine API*. This API provides convenient access to both on-device and remote sensors through the sensor datastore (see Section 5.4). The SOUL source code available is available at https://github.com/gtpcloud/SOUL.git.

5.5.1 Programming Model – SOUL Activities.

The SOUL Activity class is a Java abstract class for apps to interact with SOUL aggregates, to define its SOUL aggregates – via its *compose* method, and to finalize SOUL aggregate processing – via its *trigger* method. Trigger is important in part because sensor data processing can be specific to the applications using sensor data, an example being an application simply displaying e.g., current temperature data, vs. another that plots changes in temperature over time. The app can provide methods that plot temperature change over time, invoked by the Trigger method whenever sensor data changes. More importantly, the SOUL activity class is compatible with the Android sensor APIs because it runs as a wrapper of Android's `SensorEventListener`.

The pseudocode in Listing 5.2 shows a SOUL activity that detects intruders and turns on sirens in a home if they are not authorized to enter. The application defines its *compose()* method (line 4) to connect both the face recognition service and friendlist service associated with Facebook (line 5 – 6). It then recognizes faces in current frames of given cameras, followed by checking whether the users with those faces are authorized to enter (line 7 – 8), which results in setting a flag for the *trigger()* method (line 9). When the trigger method is invoked by the SOUL Engine, the app performs an action determined by the *intrusion* variable (line 14 - 16). It turns on all of the sirens located in the user's home. The *getService* and *getActuator* calls are explained in the next section.

5.5.2 SOUL Engine APIs

5.5.2.1 Access to Soul Aggregates

SOUL aggregates permit apps to uniformly access sensors, actuators, and associated services. If an app send a request via Android SDK to read a nearby temperature sensor, the app makes Engine API calls, for instance, the app retrieving a nearby temperature sensor makes a call in Listing 5.3.

Listing 5.2: The SOUL activity class in Android

```
1 public class IntruDetection extends SoulActivity {
2     private boolean intrusion = false;
3     @Override
4     public void compose() {
5         Soul fr = Engine.getService("FaceRecognition");
6         Soul fb = Engine.getService("Facebook");
7         if (fr.recognition(Engine.getSensors(Engine.TYPE_SENSOR_CAMERA))
8             if (!fb.getFriends.contains(fr.getRecog))
9                 intrusion = true; }
10    @Override
11    public void trigger() {
12        notifiyResult(intrusion);
13        if (intrusion) {
14            Soul siren = Engine.getActuator("siren:all", Engine.
      TYPE_LOCATION_HOME);
15            siren.turnOn();
16        }
17    }
18 }
19
```

Listing 5.3: Temperature sensor example

```
1     Engine.getSensor(Engine.TYPE_SENSOR_TEMPERATURE | Engine.
      TYPE_LOCATION_NEARBY)
```

The SOUL Engine will turn it into the JSON request shown in Listing 5.4 and send the request to the SOUL Core for access to the virtual sensor in app's aggregate. The sensor stream for delivering sensor data to the app also use the JSON format. To find such nearby sensors, SOUL utilizes PCloud managed information.

Upon success, the Engine calls *mountSensor* and if *Engine.TYPE_ACCESS_PRIVATE* is not set by the app, exports the sensor to global level via *exportSensor*. The mechanism used to notify the app about the updated sensor value is detailed in Section 5.6.1. By default, the Engine preserves the sensor's original sampling rate when raw sensor data are transmitted to the Datastore.

5.5.2.2 Group Operations

The group object for sensors is accessible via the *getSensors* function. For instance, if an app calls Android sensor APIs to use cameras that have a light-sensing capability and are available at the current location, the SOUL Engine translates such calls to

Listing 5.4: Access request to SOUL aggregate in JSON

```
{ "Soul" {
    "ops" : VS_GET,
    "sid" : 487635463,
    "sensor_name" : "_TEMPERATURE",
    "source" : "Joseph's bedroom",
    "period" : "120000", /*Sampling rate in millisec.*/
    "window" : 360 /*Time windows in sec.*/
    }
}
```

Listing 5.5: Group operations in JSON by the name of sensors

```
{ "Soul" : {
    "ops" : VS_GET,
 "flag" : VS_FLAG_SG
 "sid" : 487635464,
 "sensors" : {
     "sensor_name" : "camera",
     "sensor_name" : "light",
     . . .
     }
 }
}
```

activate an internal *groupBy* operation to sort the associated sensors with given flags to form a suitable sensor group. The group, then, provides a single point of access to multiple sensors that share a common property to trigger an internal *groupBy* operation. Internally, the SOUL Engine sends JSON request to the Core and upon response, sorts the associated sensors with given flags (e.g., source) to form a suitable sensor group, as depicted in Listing 5.5 and 5.6. The group, then, provides a single point of access to multiple sensors that share a common characteristic. That is, ensuring that the camera and light sensor are from same physical source, camera is visible from current location, etc to ensure *SensorGroup* contains proper SOUL aggregate.

5.5.2.3 Dynamic Reconfiguration

When a user's context changes, e.g., a location change, this can place a burden on apps required to explicitly deal with such changes in real time: (i) to continually check for or sense such changes, and then, (ii) discover and (iii) connect to available

Listing 5.6: Group operations in JSON with all matching sensors

```
{ "Soul" : {
    "ops" : VS_RET,
 "flag" : VS_FLAG_SG,
 "sid" : 487635464,
 "sensors" : {
     "sensor" : {
            "name" : "camera",
            "source" : "home garage cam2" },
        "sensor" : {
            "name" : "light",
            "source" : "living room cam9" },
        "sensor" : {
            "name" : "light",
            "source" : "home garage cam2"},
                . . .
        }
    }
}
```

sensors in the new context. The *mapping* method in the SOUL Engine can be used to prevent apps from having to implement such actions and experience their resulting overheads. Specifically, based on defined triggers, like a change in a user's location, a SOUL aggregate can be dynamically reconfigured to remap its sensors and actuators to newly available physical ones, without additional app intervention. To detect such context changes, SOUL utilizes the Bluetooth/GPS/WiFi along with information from PCloud .

Mapping in the Engine collects and maintains a desired characteristics of set of resources (e.g., nearby) and the Engine itself dynamically reconfigures as new sensors are discovered and old ones are no longer accessible. Addressing both sensors and actuators, mapping in the SOUL Engine defines a runtime method for binding a physical sensor to the virtual sensors in the SOUL aggregates defined by applications. Mapping also serves to bind some specific sensor processing service. The following example shows an actuator with a desired property mapped at runtime to an appropriate physical device:

Listing 5.7 shows the need to access a speaker actuator 'nearby' the device running the app, where the mapping method dynamically selects the device in question

Listing 5.7: Mapping example

```
1 Soul speaker = Engine.getActuator(``speaker");
2 speaker.mapping(Engine.TYPE_LOCATION_NEARBY);
```

at runtime among nearby available speakers. The outcome is that the app plays a sound stream via a virtual *speaker* actuator that is guaranteed to be re-mapped to speakers located nearby the device. In this way, both SOUL-aware and legacy apps using `AudioFlinger`, the Android sound system service can transparently use sensors and actuators without any overhead as the Engine continuously monitors designated sensors and actuators. The experimental evaluation in Section 5.8.6 will assess representative mapping methods and the overheads associated with using such dynamic sensors or actuators.

5.5.3 Discovering Edge Clouds

For access to edge cloud resources, SOUL seeks to discover an edge cloud whenever the user unlocks the screen on her Android device[1]. Unlocking the screen triggers an interaction with a directory service located on a remote cloud (an Amazon EC2) that returns to the device a set of edge clouds available to the user in her current environment. Which resources are returned depends on user context and her social relationships or more generally, on the access controls associated with the user requesting the resources. The *reference monitor* shown in Figure 15 implements these access controls. Their internal runtime representation is a list of access rights to services and SOUL aggregates (and the sensors they represent), with access controls enforced by SOUL's internal reference monitor. If, for example, the user drives up to her house, the reference monitor would provide full access to all in-house edge cloud resources, e.g., her home PC, home sensors, etc. If the user drives up to her friend's

[1]To reduce overheads, our current implementation limits the frequency of such discovery actions to once every five minutes.

house, such access is limited to only those resources to which her friend has granted her access, e.g., the friend's large screen display for jointly viewing vacation pictures. The FaceRecognition service used in the example above (see *getService* call) is another resource made available in this fashion, where within say, a home edge cloud, the home owner has access to and can run a rich set of services (and the resources on which they run), e.g., to determine whether a person visible on a home security camera is someone known to the homeowner or an unwanted intruder.

5.6 Select Implementation Detail

This section presents the implementation detail needed for understanding the evaluations in Section 5.8.

5.6.1 Android Realization

Our Android implementation must be (i) backwards compatible – allowing existing apps to continue to interact with their sensors as well as virtual sensors in SOUL; (ii) transparent – permitting the use of sensors regardless of their physical location, e.g., whether exist at the device or remotely; (iii) portable – allowing SOUL aggregates to run on any of the variety of edge cloud infrastructures; and (iv) controlled – enforcing well-defined access controls for the invokers of SOUL aggregates.

We obtain these properties as follows. First, SOUL aggregates wrapping existing sensors provide legacy apps with the aforementioned sensor *illusions*. Such apps can still benefit from being able to use both on-device and ambient – nearby – sensors, but cannot take advantage of more advanced SOUL functionality such as an uniform access to SOUL aggregates, nor can they bypass SOUL's access controls. Second, with sensor virtualization, i.e., the ability to construct entirely new types of sensors from both physical sensors and events (e.g., calendar events), transparency becomes a pervasive property of SOUL's Android implementation. Third, we demonstrate portability by realizing SOUL on diverse devices that include the Galaxy Prevail,

Table 7: Tested Android Devices

Device	OS	Device	OS
Nexus 7 (2013)	KitKat	Nexus 4	Kitkat
Galaxy S4	KitKat	Galaxy S3	KitKat
Galaxy Prevail	Gingerbread	Galaxy Tab 3	Jellybean

S3, S4, Tab 3, and Nexus 4, 7–2013 (see Table 7 for the detail), and by providing a translation layer with which SOUL services can run on the PCloud or elsewhere. Fourth, controlling sensor use is ensured by an access control service operated in SOUL.

The sensor framework in Android can be divided into three layers: (i) Java layer, (ii) system layer, and (iii) HAL. The Java layer interacts with apps, the system layer controls built-in sensors via HAL, and finally, HAL talks to the kernel-space device drivers. To transparently support existing apps, SOUL's implementation operates at all three layers, and for remote ambient sensors, SOUL does not make any assumption regarding how they connects to a device. (e.g., Zigbee, WiFi, bluetooth or physically connected with USB)

5.6.2 SOUL Engine

This section describes the rationale for SOUL's current Android implementation.

5.6.2.1 Hardware Abstraction Layer (HAL)

Android's HAL mandates that *when Android introduces a new sensor type that can replace an OEM-define sensor type, the OEM* ***must*** *use the official sensor type and stringType on versions of the HAL* [16]. HAL, therefore, is a seemingly attractive layer for implementing SOUL-like solutions. Unfortunately, there are several drawbacks. First, HAL itself has become **blackbox** in most Android devices, except for a handful of Google reference devices. This is because device manufacturers do not publish their HAL source codes. Second, HAL no longer operates as originally intended, e.g.,

consider the temperature sensors described in Section 5.3. As a result, apps **must** manually interact with such sensors, resulting in lack of portability and unnecessary overhead.

5.6.2.2 SensorManager in the Java layer

Recent work [37, 124] uses the Java application layer in Android to export physically attached (e.g., via USB connections) or Bluetooth-connected external sensors. This is done by providing apps with their own SDKs defined by Intent classes, using the AIDL (Android Interface Definition Language). This may help apps access on- and off-device sensors, but they must use such new SDKs, without support for the existing apps.

5.6.2.3 SensorManager in the System Layer (JNI)

Lessons from the above approaches lead us to a multi-layer approach that spans Android's system service and application framework, shown in Figure 20. (The blue boxes indicate SOUL's modules added to Android.) This multi-layer approach supports legacy apps and permits new apps written in our API to fully leverage SOUL. Android's *SensorManager* maintains (i) a *sensor list*, (ii) *sensor events*, and (iii) data about *receivers*.

getSensorList. Each Android device creates a list of its native sensors at boot time. With this list, we load both native and *dummy* sensors into *SensorDevice* at boot time, later and on demand replacing the dummies with virtual sensors. Upon request receipt, *SensorManager* returns the list of sensors available through *nativeGetNextSensor*, which simply returns the list previously loaded at boot time. In the list, *Sensor* objects contain unique Device Handler Numbers (DHNs), and to avoid overlap with the DHNs of the sensors on a device, we assign DHNs larger than 64 to virtual sensors.

SensorEvent. Apps implement *SensorEventListener* to create *SensorEventConnection* and enable hardware sensors via *addSensor* calls in *SensorEventQueue*. The queue contains a series of *SensorEvent*s. Each event contains a blocking call to retrieve data from HAL using *read(buffer, 16)* in the *handleEvent* function. Since HAL is an opaque 'blackbox', we require a mechanism for generating an event to trigger *SensorEventConnection*, for which our implementation uses a light sensor as an event 'generator' for the rate at which the fastest sensor is updated. This allows us to avoid modifying the HAL, yet still substitute the data in each event with the virtual sensor data.

Receiver. Upon receiving sensor data from the sensor storage service, SOUL on the device substitutes the data in events generated by the light sensor with incoming data, in the *handleEvent* method. In order to secure fairness for both physical and virtual sensors, we introduce a *HotQueue* variable that stores the DHNs of the next sensors to be batched. The *HotQueue* relies on Android native *property_get* and *property_set* operations, to avoid additional read/write overheads. Registered receivers in *SensorManager* wait for the blocking calls to be resolved. Since we rely on the light sensor, we not only dequeue *SensorEvent* from *SensorEventQueue*, but also dequeue DHN from *HotQueue* to compare with the DHN of the batched event, and if it has the virtual sensor's DHN, we then request an update via *updateSensor(DHN)* or *updateSensorGroup*. The result is a low latency *SensorGroup* granularity update (shown in Section 5.8.2).

5.6.3 SOUL Core

The current SOUL Core uses PCloud as the underlying edge cloud infrastructure, and PCloud offers a clean and high-level abstraction of distributed resources incldu-ing computing capabilities, sensors, and actuators. On top of PCloud , the SOUL Core implements its Sensor Datastore, access control mechanism backed by Facebook,

and resource manager to take sensor management followed by sensor-data processing from mobile devices.

5.6.3.1 Sensor Datastore

The SOUL Datastore uses OpenTSDB [13] as its underlying database. More important, however, are its actions manipulating sensor data. First, to match application-stated requirements concerning sensor data rates and at the same time, obtain high communication performance, the sensor streams it constructs 'batch' sensor data records (explained below). Second, it implements runtime reconfiguration for the mappings from physical to virtual sensors.

Batching. To efficiently transport sensor data across the network, rather than sending individual sensor records, data is batched based on two criteria: (i) the app-defined delay value (the sampling rate defined by the Android sensor framework) constitutes a constraint applied to SOUL's batching method, and (ii) networking MTU determines *batch_size* under that constraint, where *batch_size* simply represents the number of records packed into a single batch. In this fashion, SOUL adheres to common Android practice, yet obtains improved performance compared to per record network transfers. Figure 18 explains how the Datastore constructs a sensor stream with the batch optimization.

Reconfiguration. The reconfiguration service implemented by the SOUL engine (i) detects changes in the user's context, whereupon (ii) it triggers remapping between sensor streams and corresponding virtual sensors. Specifically, the current implementation detects a location change, whereupon the reconfiguration service sends a reconfiguration request to the SOUL Core along with the IDs of the sensors currently used by the application. This gives rise to a new sensor stream and ultimately, to different virtual sensor data exposed to the app. The outcome is a complete elimination of app involvement when user context is changed. The current implementation

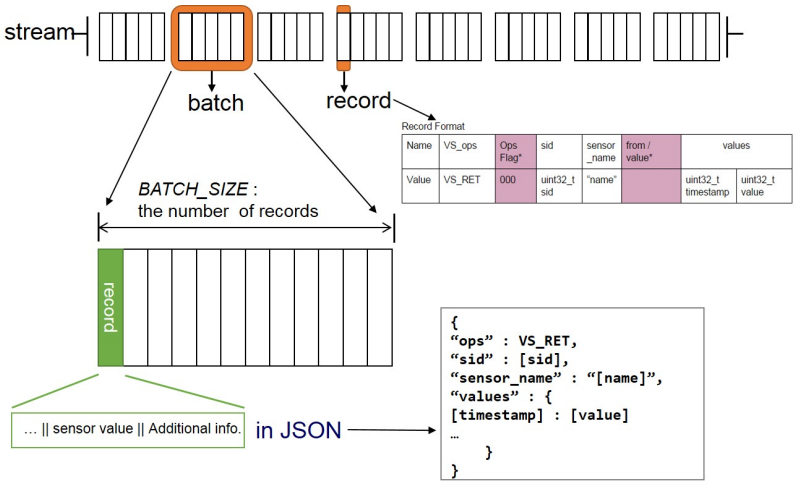


Figure 18: Batch optimization in the Datastore

does not yet consider other issues that arise when physical sensors are changed at runtime, such as potential changes in sensor resolution that may not be acceptable to an app [48].

5.6.3.2 Access Control to Mitigate Privacy Risks

To mitigate privacy and security concerns when users share sensors, we port Stencil's reference monitor to the SOUL Core. This enforces access control policies, which are inherent to how SOUL aggregates are used. As already discussed in Section 4.2, it uses the discretionary access control model for authorization and builds a Facebook-based server and app for authentication. To provide SOUL with sensor-related access control, a new operation is added as follows:

Access Control via glanceSensor. When a mobile app invokes the *openSensor* call to access shared sensors, the reference monitor returns a capability token to the app, which indicates 'no access', 'glance', or 'read'. 'No access' simply rejects such an access. The read capability allows the app to use the *getSensor* operation to fully customize requests (e.g., the time windows, filters, and resolution for data). *glanceSensor* with the glace capability is a very limited version of the *getSensor* call,

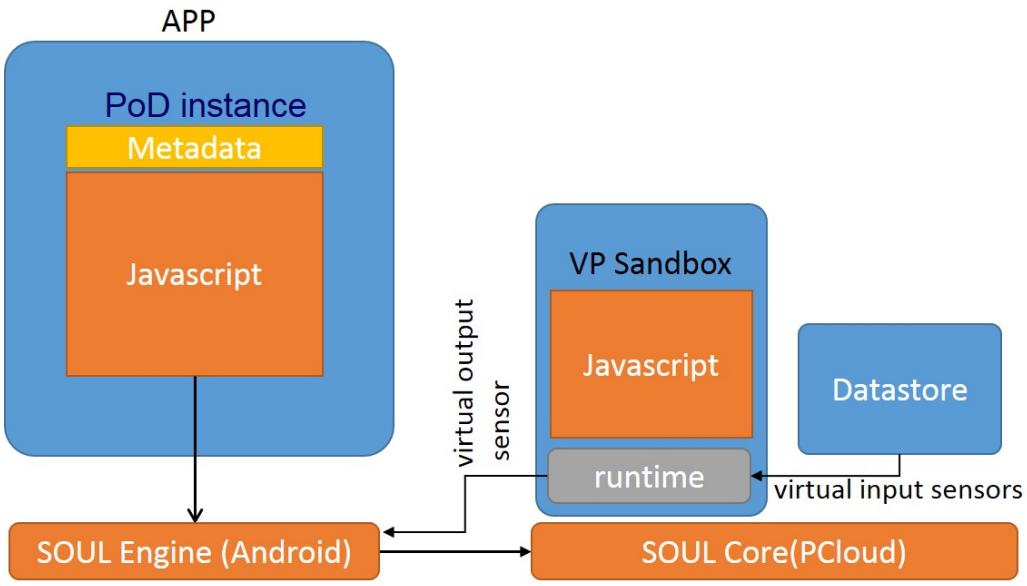


Figure 19: SOUL Processing-on-Demand

which can see only the sensor data that meets the conditions imposed by the owner-defined policy. We demonstrate this feature by implementing microaggregation [54] to the datastore, where the app with the glance capability can read only some trend of data rather than the actual raw sensor value.

5.6.3.3 Sensor Management and Processing

To permit sensor post processing methods to run anywhere, on the mobile device and/or on external resources, the resource manager can draw on PCloud resources to offload such processing 'on demand'. Toward this end, an app defines a *Processing-on-Demand* (PoD) instance, consisting of algorithm source code written in JavaScript and metadata defining PoD inputs and outputs. To run PoD code, the resource manager uses sandboxes on PCloud resources to better isolate and control their activities. The sandbox's runtime executes PoD code on Node.js and communicates with the Datastore to get and put appropriate sensor data. Processing results are again delivered to the app in form of a virtual sensor. Figure 19 depicts how an app uses external resources through its PoD instance.

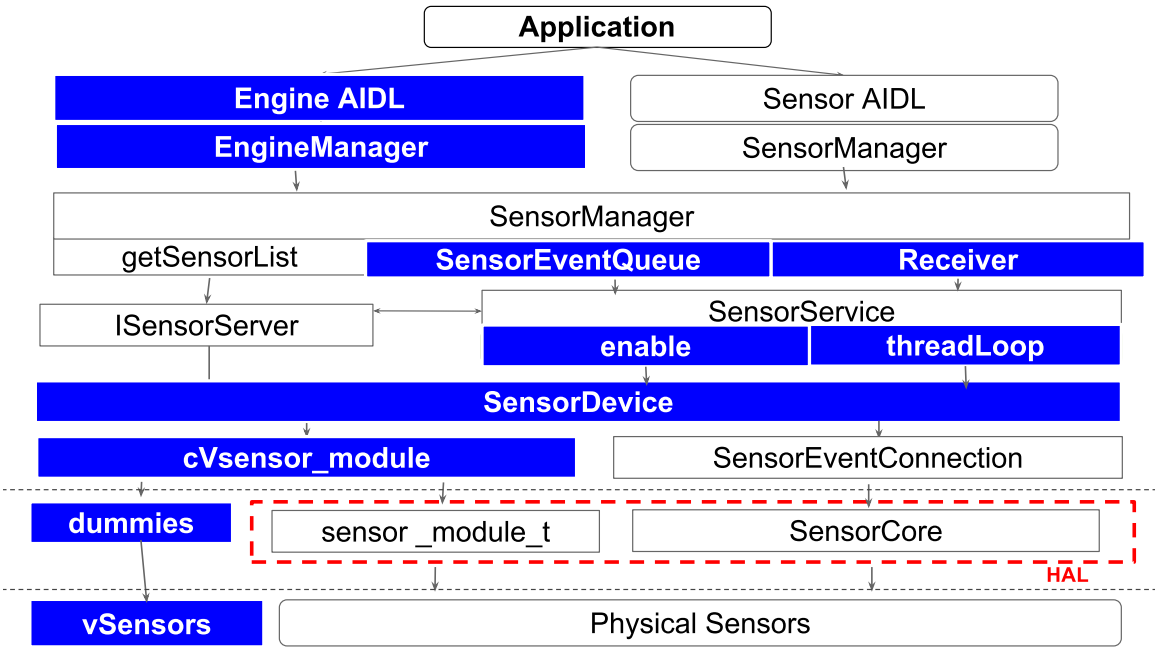


Figure 20: SOUL's implementation in Android.

5.6.3.4 Edge Cloud Translation Layer

As stated earlier, SOUL relies on an edge cloud for its use of ambient resources and the remote clouds. In SOUL, edge cloud usage is via a translation layer that acts in ways similar to say, a virtual file system in Unix. Our current implementation leverages the PCloud infrastructure described in [67], assuming the presence of a PCloud agent on any device that uses SOUL. This agent provides the single point of contact needed for initiating and finalizing SOUL execution, on the device or elsewhere, and the PCloud infrastructure with its agents provides the sandboxed environments – on ambient devices and/or the remote cloud – for running SOUL codes. We are currently investigating ports of SOUL to other edge cloud infrastructures. For instance, the Datastore would easily interact with BOLT [63], and the SOUL Core can run on top of Cloudlets with its cognitive services [65] via the translation layer.

5.7 SOUL Sensor Applications

This section describes (1) how SOUL supports and augments existing apps as well as Android internal services in their sensor use, and (2) how it presents opportunities for new kinds of apps that easily interact with sensors and use nearby and cloud resources to process their data.

5.7.1 Existing Apps & Android Services

5.7.1.1 Supporting Existing Apps using Sensors

A Google Play app, *Sensor Readout* [15], demonstrates SOUL's base support for apps to transparently and seamlessly scale in terms of numbers of sensors and their associated data processing activities. New and additional sensors can be used without modifying existing app, whether those sensors are embedded in the Android device or are accessible remotely. With SOUL, this application should not be able to tell the difference between the embedded sensors on a device and remote virtual sensors.

5.7.1.2 Augmenting Existing Apps & Services

SOUL assists apps by automatic reconfiguration of the mapping between new physical sensors and actuators to the virtual ones used by these apps. In fact, this functionality is available even to Android system services like its notification service. We demonstrate the utility of this functionality with a the novel SOUL service termed 'Everything Follows Me.', which hooks up all interactions between Android and apps & system services to create the physical-virtual sensors and actuators mapping. Along with SOUL's reconfiguration feature, this service can offer a continuous media app experience even in the case that a user's context (i.e., location) is changed without the app having to be modified.The actuators of the media app consists of an adjacent loudspeaker and screen like those present in a home media system and an LED indicator on an Intel Galileo board to forward all notifications emitted by the Android notification service. This service conducts such remapping whenever SOUL's

reconfiguration senses a user location change. The idea is to move the sound across a home's different rooms' media systems as the user walks through her home. Once the reconfiguration notifies the change of user's location to this service, it does the dynamic mapping of physical sensors and actuators to the virtual ones exposed to the application and Android system services. Using the Spotify [20] music app, with the 'Everything Follows Me' service, Spotify interacts with end users via whatever display screen and speakers are close to the user's current location. In addition, the user need not be concerned about missing out on other important Android notifications, since the 'Everything Follow me' service serves for Android internal notification service, e.g., to notify the user about an incoming text messages via the LED also present in each room.

5.7.2 Prototype SOUL Applications

An important property of the SOUL middleware is its ability to use nearby and cloud resources for potentially expensive sensor processing activities.

5.7.2.1 PoD: Processing on Demand

Raw sensor data must typically be processed for meaningful use by apps, but such processing can be expensive, quickly draining a device's battery or exceeding its processing abilities. To address this, the SOUL API permits apps to encapsulate their sensor processing codes into Javascript run by SOUL aggregates. Since it is the SOUL aggregate running these scripts, they can be run anywhere. We demonstrate this functionality with an app deploying a Kalman filter [70] to produce a statistically best estimate of sensor data: the app creates its SOUL aggregate with the filter code and executes the aggregate on edge cloud-based resources.

5.7.2.2 Composing SOUL Aggregates

A common app need is to combine and make use of multiple sensors/actuators in a uniform way to realize some desired app-level functionality. This motivates the 'sensor/actuator groups' in SOUL aggregates, where each aggregate can group and operate on multiple such groups. We demonstrate this functionality with an app that permits end users to check the current time on their smartphone, but without turning on the smartphone's battery-consuming screen. This 'Don't turn on the screen' app uses a SOUL aggregate with access to a home and smartphone camera , and a phone's speaker, along with a finger-gesture recognition software service running on the edge cloud: if the home camera sees the user approaching the smartphone, the camera triggers a camera on the smartphone to check if the user does with two fingers. Two-finger gesture is interpreted as a desire to check time vs. the user grasping the entire phone, and the response is the phone's speaker stating the current time, without unlocking and activating the screen, thus conserving phone power. Once the app defines this SOUL aggregate via the SOUL Activity class, SOUL operates the aggregate without app's interventions.

5.7.2.3 Comprehensive Health Aggregate

An additional SOUL application currently under development realizes the aforementioned 'health aggregate'. This application will allow mobile devices to become hubs for health-related information about the device owner [10, 105]. The application periodically measures data like blood sugar levels, data acquired from wearable devices (e.g., *health bracelets*), and even data acquired from say, an exercise bike used by the owner in a gym. Analytic services part of the SOUL aggregate immediately raise alarms if unusual readings are detected from the SOUL aggregate's virtual sensors, and in addition, they interact with a cloud-resident service to compute long term health statistics, implement a dashboard, etc. We offer this application as another

Table 8: SOUL Testbed Setting

Name	Hardware/Role
Sensor/Actuator Nodes	Intel Galileo 1st Gen
Camera Nodes	Exynos 5420 and AMD E450
Speaker, Monitor A/B	at room A/B respectively
EC2	m3.large (Cloud resource)
PCloud Resources	Intel i5, i7 & Core Duo
User's Device	Galaxy S4 with Kitkat(CM11)

example of one utilizing the capabilities of multiple SOUL aggregates and exercising SOUL's dynamic authorization service (e.g., when accessing the health club bicycle sensors), all driven by the SOUL engine.

5.7.2.4 Activities of Daily Living Aggregate

Another future SOUL aggregate measures the 'activities of daily home living' [82, 115], to identify changes in a subject's routine. This SOUL aggregate uses non-contact sensors to monitor subjects at home, performs computations on those sensor readings using inference algorithms, to detect erratic or abnormal behavior, and uses actuators to respond to detected abnormalities. The SOUL aggregate can be used for different applications, e.g., to diagnose dementia and diabetes by monitoring a subject's activities or to provide early warnings about seniors living at home. We offer this aggregate as an example of sensor processing requiring variably sized time series of sensor data, from multiple sensors, to determine sensor readings that are consistently abnormal. One method for processing such data uses entropy-based processing [121].

5.8 Experimental Evaluation

5.8.1 Evaluation Setup

SOUL is evaluated with micro benchmarks and with the prototype applications (apps) in Section 5.7.1. Table 8 describes our testbed, with Intel Galileo boards serving

as remote sensor and actuator nodes. Remote sensor nodes measure humidity and temperature at their locations; actuators send notifications to users via board-resident LEDs. The camera nodes track users' finger gestures for one use case and monitor surroundings. Two different physical spaces, named Room A and Room B, are each equipped with sensors, actuators, and a monitor and speaker.

5.8.2 Micro Benchmarks

5.8.2.1 Overhead

The overheads of reading SOUL's virtual vs. physical sensors are evaluated with the temperature sensor in a Samsung Galaxy S4 (GS4). In unmodified Android, such an access begins with the `SensorManager`, followed by using HAL and OEM device drivers called *SSP* in the GS4's Linux Kernel. SOUL bypasses the HAL/SSP layer, so that the `SensorManager` directly communicates with the SOUL Android implementation, as described in Section 5.6.2. Figure 21 depicts the entry and exit points of each layer for measurement. In the Figure, SM_J and SM_S denote the SensorManager in the Java (JNI) and the system layer, respectively. In Figure 22, an interesting result is that even when first storing sensor data in the Datastore and then retrieving it via the network, the app-experienced delay in SOUL is much less than that of the unmodified HAL. Even the case that SOUL core is on the remote EC2 and the device is on the relative slow 3G connection (labeled as EC2-HSPA+), SOUL shows better response time than the HAL. Figure 22 clearly shows that the time taken in the HAL is dominant in unmodified Android (99.8% of total time). With SOUL, most time is spent in the network, consequently, with nearby resources (PCloud) accessed via the 802.11g wireless LAN (WLAN) showing better performance than when using a remote cloud (EC2) accessed via T-Mobile's 3G connection(HSPA+). This result suggests that the current Android sensor HAL potentially raises huge overheads when apps access sensors. However, the limited access to the sensor HAL source code makes it very hard for us to investigate this overhead further.

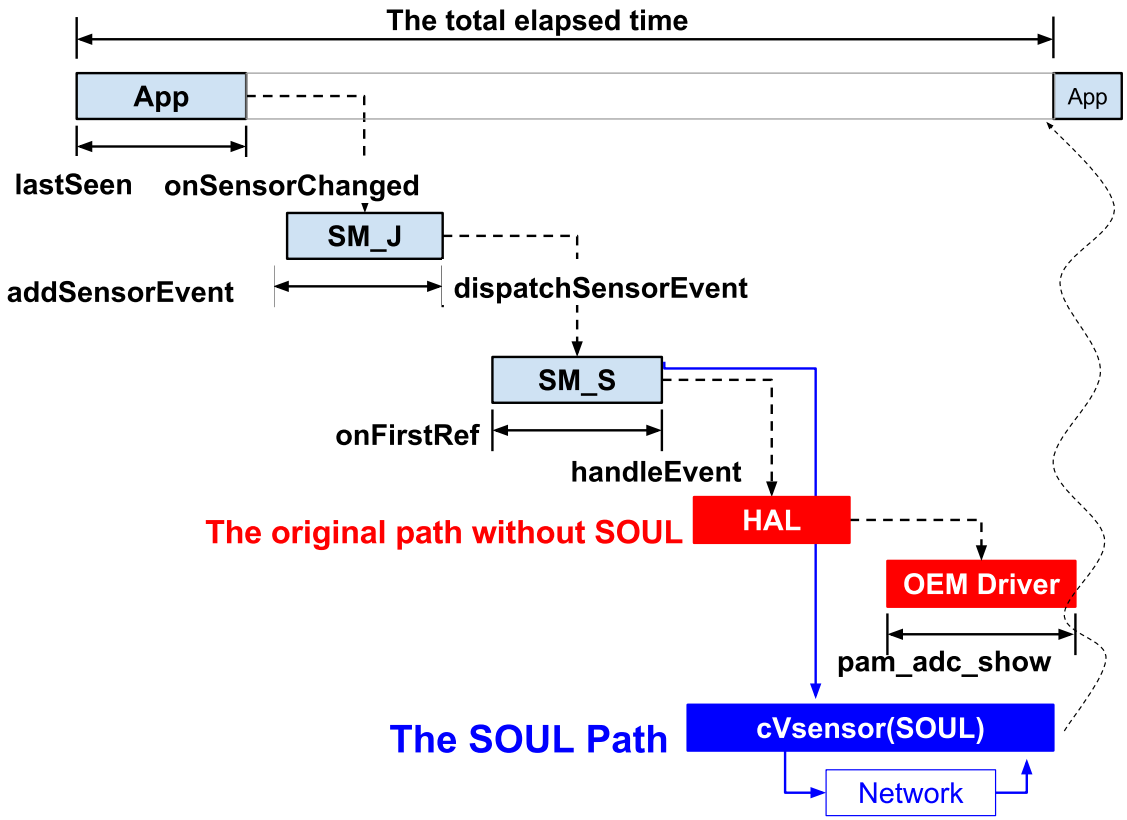


Figure 21: Measuring elapsed time per layer.

5.8.2.2 Scalability

Scalability is evaluated in terms of power consumption seen by an app, measured on a GS4 with an increasing number of sensors in one SOUL aggregate (up to 100). Power consumption is measured with a Smart Power meter [7]. To see the performance overhead when an app accesses sensors, we build a test app just reading physical sensors embedded in the device without doing any post processing. As in Figure 23, just reading five physical sensors is consuming a significant amount of CPU performance – from 20%(baseline) to 64%(Android, 5 Sensors) – resulting in dramatic increasing of CPU frequency – from 600MHz to 1.6GHz. These changes generated by the Android sensor framework can hardly be justified when an app interacts with multiple sensors

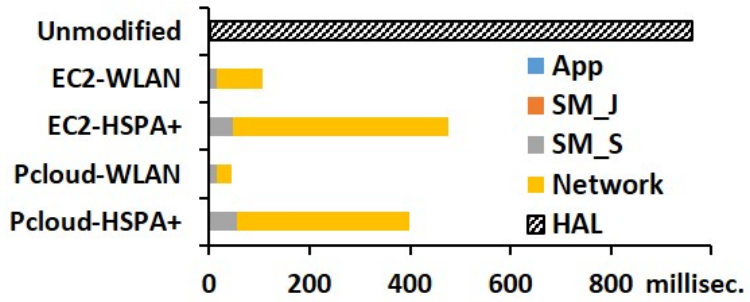


Figure 22: Elapsed time per layer.

in term of battery life. In addition to Figure 23, Figure 24(a) indicates almost constant power consumption in the SOUL case, even with increasing numbers of sensors up to 100, with the SOUL aggregate consuming less power than when a single sensor is accessed in unmodified Android (*Ref* in the figure). Latency improvements are due in part because of the 'batch' optimization (see Section 5.4.3.1) replacing many small calls with fewer larger calls when one SOUL aggregate has many virtual sensors. Figure 24(b) shows 'batch'ing gains of up to 88% in terms of latency, which results from the optimized `batch_size` when the Datastore constructs a sensor stream. Note that `batch_size` are constrained by both end user app requirements, a delay value, and the network MTU. The 'Ref' in the Figure shows the result from one sensor in unmodified Android.

5.8.3 Unmodified Apps using Sensors

Backward compatibility is evaluated by comparing sensor accesses by unmodified apps with those using SOUL. (recall that even unmodified apps can benefit from SOUL's ability to provide access to both on-device and remote sensors). We verify the backward compatibility of the SOUL by checking whether the existing apps in the Google Play Store can transparently access the physical sensors they already use via SOUL-based virtualized, an implicit bonus of such compatibility being that such sensors can be local or remote. Figure 25 is a screenshot of the Sensor Readout app able to transparently interact with on-device – the first three – and remote

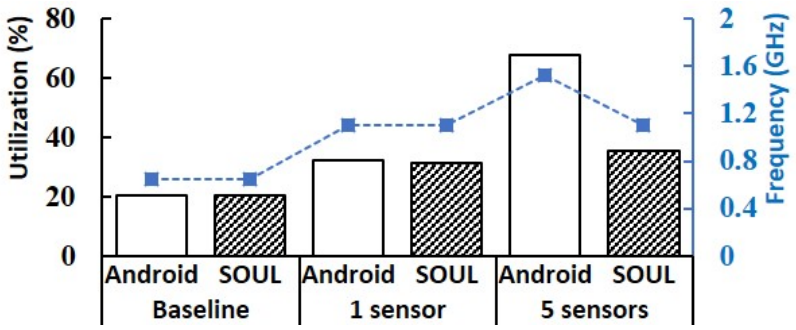


Figure 23: The CPU overhead in Android vs. SOUL

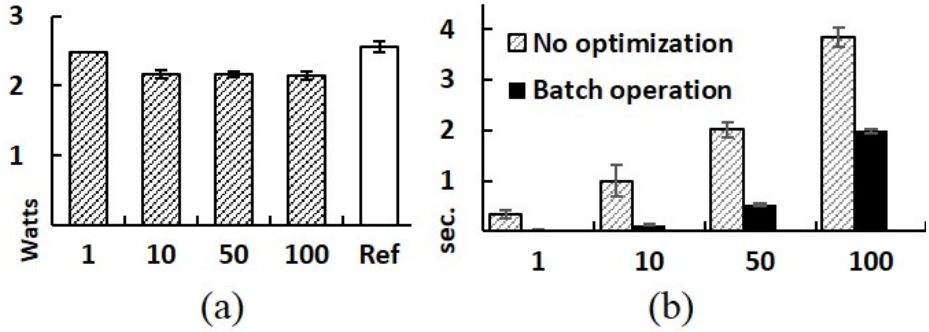


Figure 24: (a) Power consumption, and (b) Average latency.

sensors – SOUL virtual sensors. First three sensors in Figure 25 are those on our test device, GS4, while the rest are virtual sensors that are local illusions of remote sensors created by the SOUL abstraction. We also compare latency of SOUL with the native Android sensor framework when the SensorReadout app accesses a on-device sensor in Figure 26, which is very consistent with results shown in Figure 22 in Section 5.8.2. That is, avoiding the HAL layer in the Android stack is very likely to offer the better performance.

5.8.4 Processing on Demand

We use the Kalman filter to evaluate SOUL's PoD feature. Our test app creates a PoD instance with this Kalman filter source code written in JavaScript and then SOUL runs the PoD instance on a sandbox on the SOUL Core. We also run the same code on the device itself to make a comparison. Figure 27 (a) shows that it takes 1.69 seconds when such injected filter code runs in a SOUL sandbox, including all network

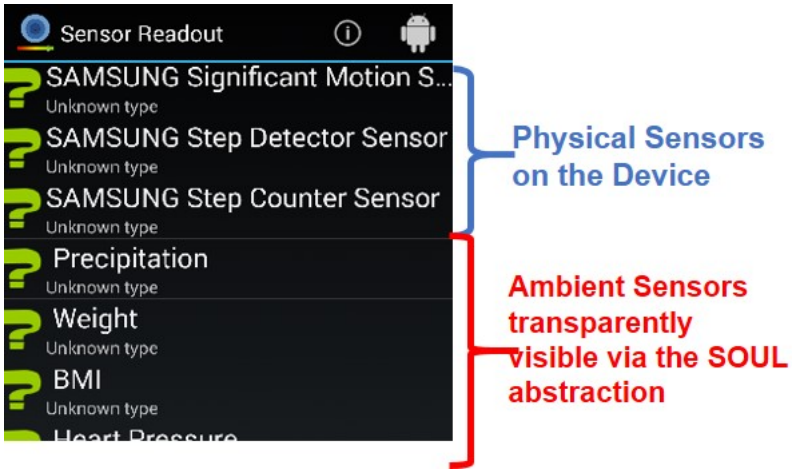


Figure 25: A Screenshot of the Sensor Readout App

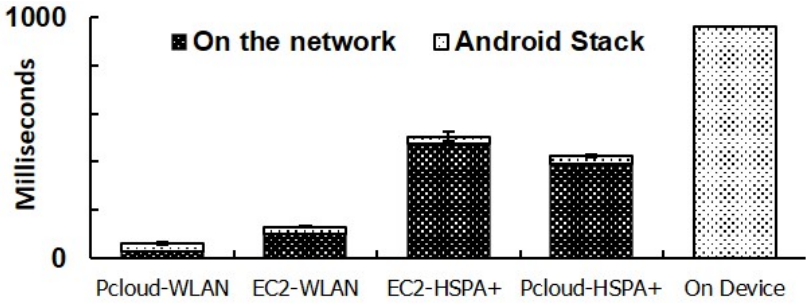


Figure 26: the Sensor Readout App-experienced delay

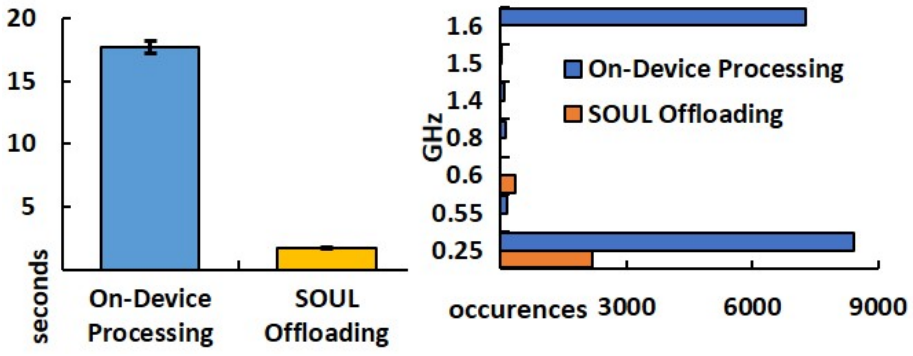
times, while it takes 17.64 seconds for the same filter processing to run on the device. There are associated gains in energy efficiency on the GS4, shown in Figure 27 (b). We counts machine cycles for each case. the filter runs 26,158,422,002 cycles on the device whereas 39,541,615 cycles on the SOUL sandbox. Reductions in elapsed time and improved energy efficiency are explained by the number of CPU-core frequency changes in Figure 27 (a), which indicates that on-device processing operates all CPU cores up to their maximum frequency (1.6GHz) for almost half of the processing time, which is very high compared to the SOUL case. Table 9 breaks the execution time into each task when its PoD instance runs on a sandbox on PCloud .

5.8.5 Composing SOUL aggregates

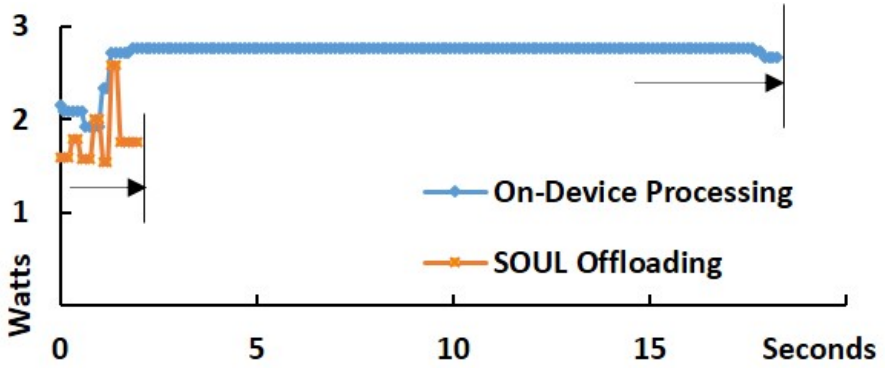
The app defines its gesture aggregate consisting of an ambient and smartphone's camera, and phone's proximity sensor as virtual sensors. The aggregate also includes phone's speaker as an actuator, and the gesture recognition service from PCloud for its post processing. The recognition service runs on nearby resources, or on the remote EC2, based on a decision made by the underlying PCloud . The end user's request – a report on the current time – is satisfied via the phone's speaker. Results are obtained by checking the current time every second for ten seconds, by either turning on the screen or via this app. Figure 29 (a) shows the elapsed time from when the finger detection service is run to when the current time is reported, with variations when the recognition service is run on local edge cloud devices vs. the remote cloud. We measure the elapsed time from the moment that the finger detection service is run to the moment that the current time is reported, when the recognition service is run on local edge cloud devices or the remote cloud. Using local resources on PCloud results in a latency of about 97.4ms (95% confidence interval: 4.8), while latency with the EC2 remote cloud is 294.5ms (30.6). We also compare the power and energy consumption of the device running this app vs. simply activating the screen and permitting the user to see the time. Figure 29 clearly shows that this app's avoidance of the screen dramatically reduces the device's energy consumption, by up to 46%. The speaking and screen consume 22.14 and 40.94 mWH, respectively. Each circle shows the moment that a user acknowledges the current time.

5.8.6 Dynamic Reconfiguration

The 'Everything Follows Me' service described in Section 5.7.1.2 provides a seamless media experience when running the Spotify app. In this evaluation, Android notifications also work with the 'Everything Follow Me' service to deliver notifications to



(a) The elapsed time and CPU frequency changes



(b) Comparison of power consumption

Figure 27: Results of the app with a Kalman filter

the nearest user-visible LED. The 'blackout' time is 1918.3 milliseconds (95% confidence interval:265.2) for the Spotify app and 10.9 milliseconds (0.21) for Android notifications. This time is the elapsed time between the moment that the SOUL's reconfiguration notifies the user's context change to the service after detecting the change of the user's location, and the moment that the services automatically remaps to available resources in the new location. This remapping happens without user or app's intervention and for an unmodified Spotify app and the Android notification service. Figure 28 (a) shows blackout time for the Android notification service and (b) for the Spotify app during reconfiguration of the SOUL object.

Table 9: PoD–Elapsed Time per task with 95% interval

Task	microsec.	95%
PoD on PCloud	946385	
– (a) Access to Datastore	18780	6708
– (b) Access Control	1530	539
– (c) Execution in Sandbox	925065	6320
Android Stack	44013	1462
Sensor stream over Network	28824	791

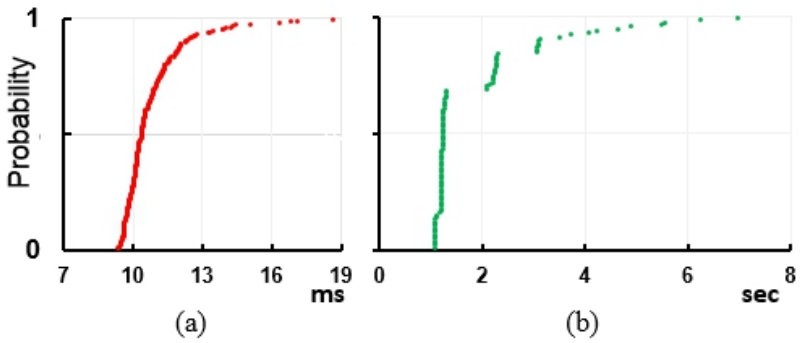


Figure 28: The CDF of 'Blackout' time.

5.8.7 Discussion

In addition to demonstrating SOUL's utility and the versatile nature of virtual sensors along with SOUL aggregates and their use, there are several interesting observations about the experimental evaluations presented above. First, offloading sensor data processing from the mobile device has advantages not only in performance and/or power consumption (e.g., consider the Kalman filter example), but also in the delays seen by end users, particularly when such offloading can use nearby computing resources vs. the remote cloud. It also presents opportunities for creating advanced functionality, like the timeseries data store used by SOUL. Second, with SOUL, even unmodified Android applications can use ambient sensors, at low cost. Third, SOUL helps create new opportunities for innovative sensing, including via sensor composition and the ability to run potentially complex processing methods on resources other than those available on a single device.

5.9 Related Work

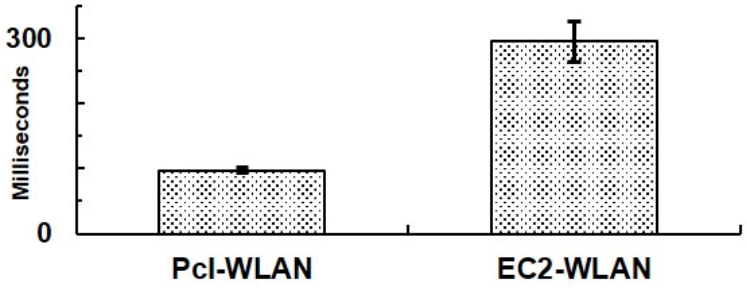
Sensor Management Frameworks in Android. The Android sensor framework [17] allows apps to access only device-resident sensors. Recent work [37, 123, 124] has suggested extensions for access to off-device sensors. ODK [37] proposes a mechanism using Kernel-level device drivers to access external sensors physically connected via USB or Bluetooth. BraceForce [124] and RTdroid [123] also do so, but because they introduce custom SDKs for interacting with external sensors, they do not support legacy apps written with the standard Android SDK. In addition, access control in RTDroid [123] is similar to what is provided by SOUL, but without SOUL's policy level support. Rio [28] virtualizes a device file interface and uses shared memory to access remote devices, potentially including sensors, but only works within single Android platforms. Finally, in contrast to sharing sparse smartphone resources across multiple entities seen in Cells [29], SOUL seeks additional flexibility in sensor processing through offloading such activities to other resources when possible.

Middleware and Programming Models The nesC [57] is a dialect of the C language for building applications for TinyOS running on sensor nodes with extremely limited hardware resources. SOUL apps run on relatively resource-rich devices that can host regular Android platforms, and SOUL can also take advantage of nearby and remote cloud resources. MiLAN [88] allows applications to define QoS properties for their sensing requirements, based on which it decides on suitable network and sensors configurations. Such techniques may be useful to further extend SOUL's policies that allocate appropriate edge, remote, and device-level resources to SOUL aggregates. MiLAN's approach may seem to be similar with the idea of virtual sensor groups in SOUL, but the purpose is completely different. While MiLAN provides different QoS levels for a specific variable of interest from physical sensors, virtual sensor groups in SOUL encapsulates multiple sensors into one virtual sensor to offer a high-level abstraction for applications via single point of access along with common

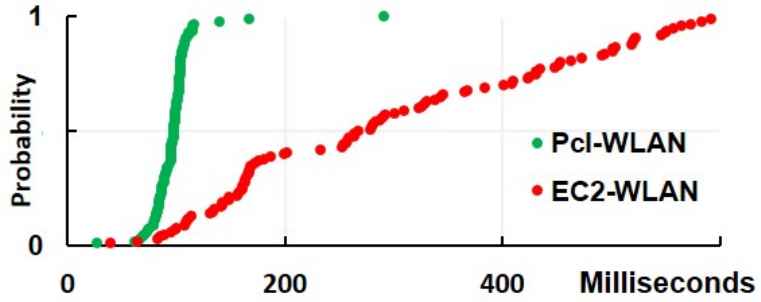
operations. SOUL allows applications to use software services offered by an edge cloud infrastructure. Also of interest may be the dynamic agent migration explored in systems like Agilla [56], which is a set of one or multiple software agents that can migrate across sensor nodes. With SOUL, however, applications are ables to provide their sensor-processing code and then SOUL allocates a best-fit resources for such code to deliver performance and energy efficiency to such applications. TeenyLIME [45] proposes a high level abstraction for data sharing among one-hop neighboring devices, but unlike SOUL aggregates, it does not fully leverage all available ambient and cloud resources. SOUL and GSN [26] share the motivation of virtual sensors, but while GSN focuses on an infrastructure for sensor network deployment and distributed query processing, SOUL provides to mobile apps new functionality that permits them to transparently and uniformly access sensors, actuators, and services, without breaking the Android programming model. Hourglass [110] introduces a *circuit* as its data model for Internet-scale sensor network and service composition. This can potentially be used for SOUL as an underlying communication layer.

Access Control and Privacy in Sensing. For SNS in a sensor-rich environment, while SOUL leverages them to enable fine-grained resource access control of sensors and associated resources, SenSocial [87] combines user activities on such services with sensing the physical context, using the user's mobile devices in a privacy-conserving manner. Hence, applications can easily capture both user context and sensed data. SOUL adopts elements of this approach. Techniques like the anonymity mechanisms in [61, 116] could be used to implement enriched SOUL's 'glance' calls, or one could use the access control-based privacy mechanism in [75]. Liu [86] suggests a new abstraction for trusted sensors with virtualization and hardware support. Obscuring data as done in statistical databases could be used to improve the Datastore. Evans et al. [53] use a Information Flow Control to manage privacy concerns in large distributed system.

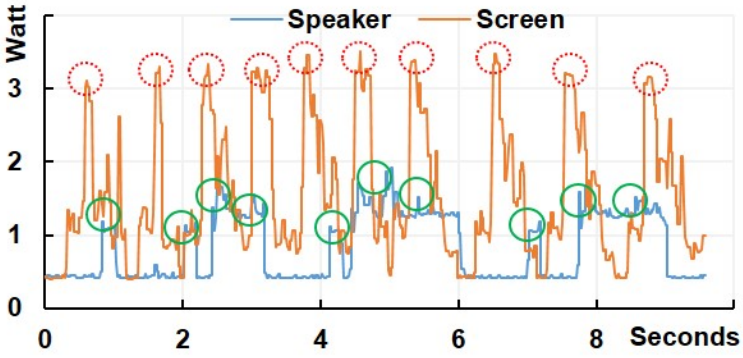
Edge Cloud Infrastructures. SOUL can be mapped to edge cloud infrastructures that include [18, 51, 64, 67], with our current work demonstrating this capability using Cloudlets. In fact, the SOUL approach is somewhat similar to recent work like BOLT [63] and Gabriel [65], both of which extend underlying cloud infrastructures with new functionalities.



(a) Response time to state the current time



(b) CDF



(c) Power consumption

Figure 29: Results of the 'Don't turn on the screen' app.

CHAPTER VI

CONCLUSION

Virtual platforms (VPs) provide applications on client devices with a high-level abstraction and an execution environment consisting of all user-accessible local and remote resources along with services and data. With virtual platforms, applications transparently use local and remote resources in a uniform way, thus benefiting from augmented capabilities beyond the limitations of a single and isolated device including lack of performance, short battery life, constrained form factors, and more importantly, the restricted scope of the data available to applications. Virtual platforms rely on a device cloud, which is a pool of network-enabled resources derived from devices at the point of use (and remote clouds), all connected via well-connected communication channels. In device clouds, one physical device exports a set of virtual resources, enabling fine-grained resource control. To realize virtual platforms as well as device clouds, three different software infrastructures were built, as follows.

STRATUS is a proof-of-concept prototype to provide applications with a best-fit set of resources in device clouds to enhance user experience without requiring applications to be changed. To this end, STRATUS constructs a VP for each user application enriched with per user *policies* to describe end user needs. Since the current implementation leverages system virtualization technology for device clouds and VPs, STRATUS assumes placement of a specific application along with its OS, dependent libraries, and user preferences into a single virtual machine (VM). That is, one VM corresponds to one application. This also makes it easier for STRATUS to avoid vendor-specific hardware or software and to preserve a user's work environment across a single machine boundary via live migration.

PCloud extends the idea of virtual platforms, mirroring lessons learned from the Stratus infrastructure. PCloud can augment device capabilities for applications to improve their performance as well as enhance the functionality seen by end users. In contrast to Stratus , PCloud can interact with non-virtualized client devices like Android smartphones. In addition to supporting non-virtualized resources, it expands its use of distributed resources that are not only from both local and personal but also from remote clouds and owned by different end users. Resource participation in PCloud is guided by permissions and policies controlled through social network services (SNS), thus making it possible to share resources owned by different end users. In addition to enriched sharing, PCloud also takes advantage of the SNS to operate its aggregation storage service, to increase the scope over which applications can operate: for common mobile applications, that scope is restricted to the data currently resident on a single device. It is interesting to note about the PCloud approach that while local resources offer low latency response, the increased data scope from remote resources can offer improved accuracy, and in addition, the substantially faster remote resources can also hide some of the additional network delay seen for requests to the remote cloud vs. local/nearby resources. Finally, replacing the policies used in Stratus , PCloud directly accepts an application's request for resources via its *Intent* API, which along with the integration of resources from clouds and other users enables VPs to more accurately understand and respond to applications' ever-changing resource requirements in a timely manner, including by scaling to the cloud resources beyond resources from a single user at the point of use.

Jointly, Soul and Stencil propose a new abstraction built on top of virtual platforms to address several issues with sensors and sensor processing ecosystem. First, Soul shields applications from today's diverse sensors and vendor-specific interfaces, thus making it easier for applications on client devices to leverage sensor data. Second, while it is highly desirable for a smartphone to act as an hub in order

to interact with the plethora of on-device and nearby sensors, the consequent sensor access and processing overheads would place undue strain on device capabilities and power consumption. SOUL aggregates, coupled with the ability to perform group- and app-specific sensor processing on resources external to the mobile device, make possible complex sensor processing and integration activities not limited by an individual smartphone's resources. Last, such applications using transient sensors visible only in certain locations or environments must dynamically acquire the rights to access and interact with them. STENCIL in SOUL provides them with required runtime permissions, including to process sensor data on nearby ambient resources and/or the remote cloud. Results evaluated in STRATUS , PCLOUD , STENCIL and SOUL indicate that virtual platforms are capable of (i) creating an abstraction of a single device comprised of distributed resources to satisfy applications' demand, (ii) providing the applications with a consistent and transparent way to access such resources available , and (iii) defining authentication, access controls via automation methods without the need for direct and repeated user interaction or consultation when the virtual platforms interact with resources, services, and data owned by different users.

Our ongoing and future work is exploring several additional dimensions of service sharing and offloading. First, we are exploring how to preserve user experience with a VP across multiple edge cloud infrastructures to support user's mobility. Second, we are experimenting with useful device-to-device interactions mediated by VPs with useful inference logic in actual home environments with less mature Internet or WiFi connectivity. In addition to the ongoing work, future work will continue to experiment with and devise new and innovative applications, and use both to further refine VP abstractions and their functionalities.

REFERENCES

[1] "Architectural requirements for always-on subsystems." http://goo.gl/MozVwY.

[2] "Dropbox." http://www.dropbox.com.

[3] "Google app engine." http://code.google.com/appengine.

[4] "Google play apps crawler." http://goo.gl/qt4Q5K.

[5] "Intel(r) perceptual computing sdk documentation." http://goo.gl/v6C87G.

[6] "Microsoft's azure cloud suffers first crash." http://www.theregister.co.uk/2009/03/16/azure-cloud-crash.

[7] "Odroid smart power." http://goo.gl/aGLyaJ.

[8] "Open sensor platform." http://www.sensorplatforms.com/osp/.

[9] "Playdrone." http://goo.gl/15RaCS.

[10] "The potential of sensor-based monitoring as a tool for health care, health promotion, and research," *Ann Fam Med*, vol. 9, no. 4, pp. 296 – 298.

[11] "Press release." http://goo.gl/CziKJa.

[12] "The remote framebuffer protocol." http://tools.ietf.org/html/rfc6143.

[13] "The scalable time series database." http://opentsdb.net/.

[14] "Sensor-based apps offer lots of potential but are hindered by fragmentation." http://goo.gl/aUhi8N.

[15] "Sensor readout." http://goo.gl/yYiZhP.

[16] "sensor.h." http://goo.gl/U2tVRX.

[17] "Sensors overview." http://goo.gl/rSVKRD.

[18] "Smartthings." https://www.smartthings.com.

[19] "Sparkleshare." http://sparkleshare.org.

[20] "Spotify." https://www.spotify.com/us/.

[21] "Sun ray." http://www.oracle.com/us/technologies/virtualization/oraclevm/061984.html.

[22] "The t-mobile-microsoft sidekick data disaster: Poor it management going mainstream." http://www.zdnet.com/blog/btl/the-t-mobile-microsoft-sidekick-data-disaster-poor-it-management-going-mainstream/25777.

[23] "Windows azure." http://www.microsoft.com/windowsazure.

[24] "The wireless communications alliance 2012." http://goo.gl/fs7hUq.

[25] "Zeromq." http://zeromq.org.

[26] Aberer, K., Hauswirth, M., and Salehi, A., "Infrastructure for data processing in large-scale interconnected sensor networks," in *Mobile Data Management, 2007 International Conference on*, pp. 198–205, May 2007.

[27] Aditya, P., Erdélyi, V., Lentz, M., Shi, E., Bhattacharjee, B., and Druschel, P., "Encore: Private, context-based communication for mobile social apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 135–148, ACM, 2014.

[28] Amiri Sani, A., Boos, K., Yun, M. H., and Zhong, L., "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of Mobisys*, MobiSys '14, (New York, NY, USA), pp. 259–272, ACM, 2014.

[29] Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J., "Cells: A virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 173–187, ACM, 2011.

[30] Annamalai, M., Birrell, A., Fetterly, D., and Wobber, T., "Implementing portable desktops: a new option and comparisons," Tech. Rep. TR-2006-151, Microsoft Research, October 2006.

[31] Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H.-I., "The case for cyber foraging," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), pp. 87–92, ACM, 2002.

[32] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[33] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[34] Bauer, D., Blough, D. M., and Cash, D., "Minimal information disclosure with efficiently verifiable credentials," in *Proceedings of the 4th ACM Workshop on Digital Identity Management*, DIM '08, (New York, NY, USA), pp. 15–24, ACM, 2008.

[35] Beach, A., Gartrell, M., Xing, X., Han, R., Lv, Q., Mishra, S., and Seada, K., "Fusing mobile, sensor, and social data to fully enable context-aware computing," in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, HotMobile '10, (New York, NY, USA), pp. 60–65, ACM, 2010.

[36] Bhardwaj, K., Sreepathy, S., Gavrilovska, A., and Schwan, K., "Ecc: Edge cloud composites," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pp. 38–47, April 2014.

[37] Brunette, W., Sodt, R., Chaudhri, R., Goel, M., Falcone, M., Van Orden, J., and Borriello, G., "Open data kit sensors: A sensor integration framework for android at the application-level," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, (New York, NY, USA), pp. 351–364, ACM, 2012.

[38] Brush, A. B., Jung, J., Mahajan, R., and Martinez, F., "Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors," in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, (New York, NY, USA), pp. 693–700, ACM, 2013.

[39] Buevich, M., Wright, A., Sargent, R., and Rowe, A., "Respawn: A distributed multi-resolution time-series datastore," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pp. 288–297, Dec 2013.

[40] Cáceres, R., Carter, C., Narayanaswami, C., and Raghunath, M., "Reincarnating pcs with portable soulpads," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, (New York, NY, USA), pp. 65–78, ACM, 2005.

[41] Chandra, R., Zeldovich, N., Sapuntzakis, C., and Lam, M. S., "The collective: A cache-based system management architecture," in *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, (Berkeley, CA, USA), pp. 259–272, USENIX Association, 2005.

[42] Chu, C.-T., Jung, J., Liu, Z., and Mahajan, R., "strack: Secure tracking in community surveillance," in *Proceedings of the ACM International Conference on Multimedia*, MM '14, (New York, NY, USA), pp. 837–840, ACM, 2014.

[43] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A., "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 301–314, ACM, 2011.

[44] Cidon, A., London, T. M., Katti, S., Kozyrakis, C., and Rosenblum, M., "Mars: adaptive remote execution for multi-threaded mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, MobiHeld '11, (New York, NY, USA), pp. 1:1–1:6, ACM, 2011.

[45] Costa, P., Mottola, L., Murphy, A. L., and Picco, G. P., "Programming wireless sensor networks with the teenylime middleware," in *Proceedings of Middleware*, Middleware '07, (New York, NY, USA), pp. 429–449, Springer-Verlag New York, Inc., 2007.

[46] Crowley, J., "Principles and techniques for sensor data fusion," in *Multisensor Fusion for Computer Vision* (Aggarwal, J., ed.), vol. 99 of *NATO ASI Series*, pp. 15–36, Springer Berlin Heidelberg, 1993.

[47] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P., "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, (New York, NY, USA), pp. 49–62, ACM, 2010.

[48] Das, T., Mohan, P., Padmanabhan, V. N., Ramjee, R., and Sharma, A., "Prism: Platform for remote sensing using smartphones," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, (New York, NY, USA), pp. 63–76, ACM, 2010.

[49] Dearman, D. and Pierce, J. S., "It's on my other computer!: computing with multiple devices," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, (New York, NY, USA), pp. 767–776, ACM, 2008.

[50] Dearman, D. and Pierce, J. S., "It's on my other computer!: Computing with multiple devices," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, (New York, NY, USA), pp. 767–776, ACM, 2008.

[51] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, P., "An operating system for the home," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2012.

[52] Edwards, W. K., Newman, M. W., Sedivy, J. Z., Smith, T. F., Balfanz, D., Smetters, D. K., Wong, H. C., and Izadi, S., "Using speakeasy

for ad hoc peer-to-peer collaboration," in *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, (New York, NY, USA), pp. 256–265, ACM, 2002.

[53] Evans, D., Eyers, D. M., and Bacon, J., "A model of information flow control to determine whether malfunctions cause the privacy invasion," in *Proceedings of MPM*, MPM '12, (New York, NY, USA), pp. 2:1–2:6, ACM, 2012.

[54] Fayyoumi, E. and Oommen, B. J., "A survey on statistical disclosure control and micro-aggregation techniques for secure statistical databases," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1161–1188, 2010.

[55] Flinn, J. and Satyanarayanan, M., "Energy-aware adaptation for mobile applications," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, (New York, NY, USA), pp. 48–63, ACM, 1999.

[56] Fok, C.-L., Roman, G.-C., and Lu, C., "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 16:1–16:26, July 2009.

[57] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D., "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 1–11, ACM, 2003.

[58] Gilbert, E., *Computing Tie Strength*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.

[59] Gilbert, E., "Predicting tie strength in a new medium," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, (New York, NY, USA), pp. 1047–1056, ACM, 2012.

[60] Gilbert, E. and Karahalios, K., "Predicting tie strength with social media," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, (New York, NY, USA), pp. 211–220, ACM, 2009.

[61] Gruteser, M., Schelle, G., Jain, A., Han, R., and Grunwald, D., "Privacy-aware location sensor networks," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2003.

[62] Guillaumin, M., Verbeek, J., and Schmid, C., "Is that you? metric learning approaches for face identification," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 498–505, Sept 2009.

[63] Gupta, T., Singh, R. P., Phanishayee, A., Jung, J., and Mahajan, R., "Bolt: Data management for connected homes," in *Proceedings of NSDI*, USENIX, April 2014.

[64] Ha, K., Chen, Z., Hu, W., Richter, W., Pillai, P., and Satyanarayanan, M., "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 68–81, ACM, 2014.

[65] Ha, K., Chen, Z., Hu, W., Richter, W., Pillai, P., and Satyanarayanan, M., "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 68–81, ACM, 2014.

[66] Jang, M. and Schwan, K., "Stratus: Assembling virtual platforms from device clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 476–483, 2011.

[67] Jang, M., Schwan, K., Bhardwaj, K., Gavrilovska, A., and Avasthi, A., "Personal clouds: Sharing and integrating networked resources to enhance end user experiences," in *INFOCOM, 2014 Proceedings IEEE*, pp. 2220–2228, April 2014.

[68] Jang, M. and Schwan, K., "Stratus: Assembling virtual platforms from device clouds," *2013 IEEE Sixth International Conference on Cloud Computing*, vol. 0, pp. 476–483, 2011.

[69] Jenks, G. F., "The data model concept in statistical mapping," in *International Yearbook of Cartography*, pp. 186–190, 2004.

[70] Kalman, R. E., "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[71] Kemp, R., Palmer, N., Kielmann, T., and Bal, H., "Cuckoo: A computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services* (Gris, M. and Yang, G., eds.), vol. 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 59–79, Springer Berlin Heidelberg, 2012.

[72] Kirby, M. and Sirovich, L., "Application of the karhunen-loeve procedure for the characterization of human faces," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 12, pp. 103–108, Jan 1990.

[73] Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X., "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *INFOCOM, 2012 Proceedings IEEE*, pp. 945–953, 2012.

[74] Kozuch, M. and Satyanarayanan, M., "Internet suspend/resume," in *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pp. 40–46, 2002.

[75] Kulkarni, D. and Tripathi, A., "Context-aware role-based access control in pervasive computing systems," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, (New York, NY, USA), pp. 113–122, ACM, 2008.

[76] Kumar, K. and Lu, Y.-H., "Cloud computing for mobile users: Can offloading computation save energy?," *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[77] Kumar, S., Agarwala, I., and Schwan, K., "Netbus: A transparent mechanism for remote device access in virtualized systems," tech. rep., 2008.

[78] Kumar, S. and Schwan, K., "Netchannel: A vmm-level mechanism for continuous, transparentdevice access during vm migration," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, (New York, NY, USA), pp. 31–40, ACM, 2008.

[79] Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K., "vmanage: loosely coupled platform and virtualization management in data centers," in *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, (New York, NY, USA), pp. 127–136, ACM, 2009.

[80] Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K., "vmanage: Loosely coupled platform and virtualization management in data centers," in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, (New York, NY, USA), pp. 127–136, ACM, 2009.

[81] Kumar, S., Talwar, V., Ranganathan, P., Nathuji, R., and Schwan, K., "M-channels and m-brokers: Coordinated management in virtualized systems."

[82] Lee, B. and Kim, H., "A design of context aware smart home safety management using by networked rfid and sensor," in *Home Networking* (Al Agha, K., Carcelle, X., and Pujolle, G., eds.), vol. 256 of *IFIP The International Federation for Information Processing*, pp. 215–224, Springer US, 2008.

[83] Lee, S.-M., Suh, S.-B., Jeong, B., Mo, S., Jung, B. M., Yoo, J.-H., Ryu, J.-M., and Lee, D.-H., "Fine-grained i/o access control of the mobile devices based on the xen architecture," in *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, (New York, NY, USA), pp. 273–284, ACM, 2009.

[84] Li, H. P., Hu, H., and Xu, J., "Nearby friend alert: Location anonymity in mobile geosocial networks," *Pervasive Computing, IEEE*, vol. 12, pp. 62–70, Oct 2013.

[85] Liu, B., Jiang, Y., Sha, F., and Govindan, R., "Cloud-enabled privacy-preserving collaborative learning for mobile sensing," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, (New York, NY, USA), pp. 57–70, ACM, 2012.

[86] Liu, H., Saroiu, S., Wolman, A., and Raj, H., "Software abstractions for trusted sensors," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, (New York, NY, USA), pp. 365–378, ACM, 2012.

[87] Mehrotra, A., Pejovic, V., and Musolesi, M., "Sensocial: A middleware for integrating online social networks and mobile sensing data streams," in *Proceedings of Middleware*, Middleware '14, (New York, NY, USA), pp. 205–216, ACM, 2014.

[88] Murphy, A. L. and Heinzelman, W. B., "Milan: Middleware linking applications and networks," tech. rep., Rochester, NY, USA, 2002.

[89] Nakazawa, J., Tokuda, H., Edwards, W., and Ramachandran, U., "A bridging framework for universal interoperability in pervasive systems," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pp. 3–3, 2006.

[90] Osman, S., Subhraveti, D., Su, G., and Nieh, J., "The design and implementation of zap: A system for migrating computing environments," in *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002*, pp. 361–376, 2002.

[91] Oulasvirta, A. and Sumari, L., "Mobile kits and laptop trays: Managing multiple devices in mobile information work," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, (New York, NY, USA), pp. 1127–1136, ACM, 2007.

[92] Oulasvirta, A. and Sumari, L., "Mobile kits and laptop trays: Managing multiple devices in mobile information work," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, (New York, NY, USA), pp. 1127–1136, ACM, 2007.

[93] Pierce, J. S. and Nichols, J., "An infrastructure for extending applications' user experiences across multiple personal devices," in *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, (New York, NY, USA), pp. 101–110, ACM, 2008.

[94] Pierce, J. S. and Nichols, J., "An infrastructure for extending applications' user experiences across multiple personal devices," in *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, (New York, NY, USA), pp. 101–110, ACM, 2008.

[95] Poellabauer, C., Abbasi, H., and Schwan, K., "Cooperative run-time management of adaptive applications and distributed resources," in *Proceedings of the Tenth ACM International Conference on Multimedia*, MULTIMEDIA '02, (New York, NY, USA), pp. 402–411, ACM, 2002.

[96] Qiu, L., Zhang, Y., Wang, F., Kyung, M., and Mahajan, H. R., "Trusted computer system evaluation criteria," in *National Computer Security Center*, 1985.

[97] Rahimi, M., Venkatasubramanian, N., Mehrotra, S., and Vasilakos, A., "Mapcloud: Mobile applications on an elastic and scalable 2-tier cloud architecture," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pp. 83–90, 2012.

[98] Raj, H., Seshasayee, B., O'Hara, K. J., Nathuji, R., Schwan, K., and Balch, T., "Spirits: Using virtualization and pervasiveness to manage mobile robot software systems," in *Proceedings of the Second IEEE International Conference on Self-Managed Networks, Systems, and Services*, SelfMan'06, (Berlin, Heidelberg), pp. 116–129, Springer-Verlag, 2006.

[99] Ramachandran, A. V. and Feamster, N., "Authenticated out-of-band communication over social links," in *Proceedings of the First Workshop on Online Social Networks*, WOSN '08, (New York, NY, USA), pp. 61–66, ACM, 2008.

[100] Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E., "Knit: Component composition for systems software," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, (Berkeley, CA, USA), pp. 24–24, USENIX Association, 2000.

[101] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H., and Cowan, C., "User-driven access control: Rethinking permission granting in modern operating systems," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 224–238, May 2012.

[102] Rolia, J., Cherkasova, L., Arlitt, M., and Andrzejak, A., "A capacity management service for resource pools," in *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, (New York, NY, USA), pp. 229–237, ACM, 2005.

[103] Román, M. and Campbell, R. H., "A middleware-based application framework for active space applications," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, (New York, NY, USA), pp. 433–454, Springer-Verlag New York, Inc., 2003.

[104] Rosu, D., Schwan, K., Yalamanchili, S., and Jha, R., "On adaptive resource allocation for complex real-time applications," in *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pp. 320–329, Dec 1997.

[105] Sakir Uen, Rolf Fimmers, M. B. G. N. and Mengden, T.

[106] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N., "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, 2009.

[107] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N., "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, pp. 14–23, Oct. 2009.

[108] Schmidt, B. K., Lam, M. S., and Northcutt, J. D., "The interactive performance of slim: A stateless, thin-client architecture," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, (New York, NY, USA), pp. 32–47, ACM, 1999.

[109] Shankar, P., Nath, B., Iftode, L., Ananthanarayanan, V., and Han, L., "Sbone: Personal device sharing using social networks," 2010.

[110] Shneidman, J., Pietzuch, P., Ledlie, J., Roussopoulos, M., Seltzer, M., and Welsh, M., "Hourglass: An infrastructure for connecting sensor networks and applications," tech. rep., Harvard University TR-21-04, Cambridge, MA, 2004.

[111] Silva, L. C. D., Morikawa, C., and Petra, I. M., "State of the art of smart homes," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 7, pp. 1313 – 1321, 2012. Advanced issues in Artificial Intelligence and Pattern Recognition for Intelligent Surveillance System in Smart Home Environment.

[112] Song, X. and Ramachandran, U., "Mobigo: A middleware for seamless mobility," in *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pp. 249–256, Aug 2007.

[113] Soror, A. A., Minhas, U. F., Aboulnaga, A., Salem, K., Kokosielis, P., and Kamath, S., "Automatic virtual machine configuration for database workloads," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 953–966, ACM, 2008.

[114] Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J., "A feedback-driven proportion allocator for real-rate scheduling," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 145–158, USENIX Association, 1999.

[115] Stone, E. and Skubic, M., "Evaluation of an inexpensive depth camera for in-home gait assessment," *J. Ambient Intell. Smart Environ.*, vol. 3, pp. 349–361, Dec. 2011.

[116] Sweeney, L., "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, pp. 557–570, Oct. 2002.

[117] T. Matthews, J. P. and Tang, J., "No smart phone is an island: the impact of places, situations, and other devices on smart phone use," Tech. Rep. RJ10452, IBM, 2009.

[118] Thrun, S., "Toward robotic cars," *Commun. ACM*, vol. 53, pp. 99–106, Apr. 2010.

[119] Unisys, "Unisys security index," 2009.

[120] Viennot, N., Garcia, E., and Nieh, J., "A measurement study of google play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, (New York, NY, USA), pp. 221–233, ACM, 2014.

[121] Wang, C., Talwar, V., Schwan, K., and Ranganathan, P., "Online detection of utility cloud anomalies using metric distributions," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pp. 96–103, April 2010.

[122] Want, R., Pering, T., Sud, S., and Rosario, B., "Dynamic composable computing," in *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications*, HotMobile '08, (New York, NY, USA), pp. 17–21, ACM, 2008.

[123] Yan, Y., Cosgrove, S., Anand, V., Kulkarni, A., Konduri, S. H., Ko, S. Y., and Ziarek, L., "Real-time android with rtdroid," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 273–286, ACM, 2014.

[124] Zheng, X., Perry, D. E., and Julien, C., "Braceforce: A middleware to enable sensing integration in mobile applications for novice programmers," in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2014, (New York, NY, USA), pp. 8–17, ACM, 2014.

VITA

Minsung Jang is a Ph.D. candidate in Computer Science at the Georgia Institute of Technology, advised by Dr. Karsten Schwan. His current research concerns software methods and platforms that promote the seamless integration of edge devices with each other and with private or public clouds. He is also interested in virtualization technologies for mobile cloud computing with workload offloading and the virtual machine migration. More generally, his interests are in experimental systems, ranging from working with hypervisors, to operating systems, to innovative middleware and applications, with past work including contributions to multimedia file systems, real-time embedded Linux systems, and the XenARM project at Software Laboratories of Samsung Electronics, and Sindo Ricoh. Minsung obtained his undergraduate and master's degrees from Yonsei University, Korea.