Imperial College London

Department of Computing

Hive: An Extensible and Scalable Framework For Mobile Crowdsourcing

Author:
Dimitar Valentinov Pavlov

Supervisor:
Dr. Emil Lupu

Submitted in partial fulfillment of the requirements
for the MSc Degree in Advanced Computing of Imperial College London

September 6, 2013

Abstract

Mobile crowdsourcing (MCS) is a fast-growing field with many applications which are due to the rapid adoption, ubiquity and powerful capabilities of current smartphones and other mobile devices. While the applications are various and novel, their building blocks can often be reduced to common operations involving data handling, user interactions, mobile or server analytics, and managing user participation. The development of a system that performs such actions is time-consuming and can be difficult to address the unique MCS challenges given that the parties interested in MCS can be diverse and with different backgrounds. This project offers a general software framework, Hive, which allows its users to focus working only on the novel aspects of the mobile crowdsourcing application without needing to reinvent the wheel for any of the common functionality. Hive makes it easy to specify new MCS tasks with quite reduced development effort, or in some cases — none at all. There is only one other framework (Medusa [25]) with similar to Hive's goals but it has its limitations. The lack of such solutions in the literature has reinforced the motivation behind this project.

Hive enables non-technical MCS task specification and the reuse of already implemented solutions in new tasks. What is specific about Hive is that it is oriented towards the support of third-party developers who can easily extend its functionality — a practical yet a demanding feature to realize that ensures the framework's future generality and extensibility. The presented system is shown to be more succinct and requiring less technical expertise than Medusa in the declaration of new MCS tasks — something which Medusa is already quite successful at when compared to ad-hoc MCS applications. Furthermore, the design choices made for Hive make it more scalable and create the foundation for the specification of general tasks — ones that are not only done on the mobile device but in fact interleave MCS with cloud processing, such as training machine learning services. Also, because of the way the system concerns are separated between the client and the server, the little dependence the client has of the server when it comes to task execution, the client's robust performance, and the close to real-time client notifications, Hive enables the execution of tasks which would not be possible when using Medusa. Lastly, the design and implementation provided for a part of the framework — the mobile client, result in a novel mobile application which acts as a framework to other applications built by third-party developers and yet controlled by the framework application. This design may be adapted for other, non-MCS related topics that also require mobile applications acting as frameworks.

Contents

List of Figures

Chapter 1

Introduction

In the past five years the computing industry has been going through a trend which many now label as the beginning of the post-PC era. The term refers to the behavioural shift from using of a single computing device, often a desktop one, per person to a lifestyle of mixing different devices of various forms and capacity. The proliferation of rich web services and cloud computing, advancements in battery capacity and CPU energy efficiency, adoption of new UI interaction patterns and the availability of cheap storage as an enabler for Big Data machine learning have altogether contributed to the formation of an environment in which people rely on several computing devices depending on the context they are in. In such environment, smartphones and tablets are thriving.

Equipped with various sensors – gyroscope, GPS, camera, digital compass, accelerometer, microphone, light and proximity sensors – smartphones are not only excellent communication and computation devices but they are also very suitable for ubiquitous computing use cases. Through their mobility and ubiquity, smartphones can act as an enabler for large-scale sensing applications which in the past would have been only ad-hoc and would have incurred a much larger deployment cost due to the need of expensive specialized hardware.

What is more, the numbers of smartphone owners is growing at a tremendous rate. The precise number is not publicly available but can be approximated from the number of smartphones sold. According to analysts, more than 700 million smartphones were sold worldwide in 2012, a 44 per cent increase compared to 2011 [12], [1]. 2013 is said to be the year when more than a billion smartphones are sold [9] with 70 to 80 percent of the annual growth attributed to developing countries [4] — places where technology proliferation has historically been very difficult due to lacking infrastructure and high costs. This statistic alone points to the rapid increase in new users.

With these observations it is reasonable to say that smartphones – through both their ubiquity and growing pervasiveness – can be a great opportunity to impact people's lives on an unprecedented in the computing history scale.

1.1 Project Motivation

The accelerated improvement and adoption of smartphones is in the process of creating new industries to work in and opportunities to research and innovate. One such domain is mobile crowdsourcing. It leverages the intelligence of large amounts of ordinary smartphone users to obtain information. As a complement to crowdsourcing, mobile crowdsensing seeks to use the data available from the multiple sensors present on modern smartphones. A significant challenge is assessing the data validity and whether it is still relevant (i.e., up-to-date). Both mobile crowdsensing and crowdsourcing are closely related topics and they will be collectively referred to as MCS. As the literature review will show, MCS poses new challenges and the field is still in its early stages of development. A common approach to evaluating MCS research is the development of a MCS system for the particular application and assessing how well the proposed combination of sensor data and/or user input solves the particular problem. Such problems can be obtaining real-time traffic conditions and pothole locations [23], joint control of a robot's interface [18] or receiving a real-time instructive assistance for blind people [3].

However, it is still the case that for any new MCS task, a new system is built from the very beginning. This often includes a mobile application (client) built for at least one mobile operating system and a server back-end. This ad hoc architecture approach creates several issues:

- The software built is usually a proof-of-concept and is only a means for evaluation of the proposed approach. Yet, the system design challenges and development effort involved may be a barrier for conducting the actual research of how MCS can be used to solve the given problem. In the very least, they will slow down the research work significantly or affect the quality of the evaluation.
- Some of the key problems characterizing MCS may not be addressed (for example, due to time constraints) if they are not the focus of the research. Such problems are privacy, allowing different levels of human involvement as well as the impact on the normal use of the phone — battery drain, phone responsiveness, and blocked sensors for normal uses. However, addressing these problems is important for any real solution that is to be deployed.
- There is a good amount of functionality overlap between different MCS applications — all of them involve some form of data collection, processing, and storage which require specific kinds of infrastructure and services. In addition, the data collected itself may be relevant to more than one application. However, the above ad hoc approach does not allow for any reuse of any created solutions or straightforward data sharing.
- For the large scale adoption that is needed for MCS to become useful, installing a mobile app for every new problem is impractical and stifles the user involvement. This in turn further reduces the usefulness of MCS. A large-scale involvement is especially important for responsiveness close to real-time and data collection for machine learning problems.

In parallel to the work done in the field of MCS, the machine learning (ML) community has been developing algorithms designed for crowdsourced data. Its main purpose has been obtaining ground truth from the data at minimum crowd cost (number of people asked to perform the same task). The literature review for this project has identified a lack of research at the intersection of between crowdsourced machine learning algorithms and MCS. A hybrid approach between the two areas could be quite beneficial — if the ML model is stuck or produces output with low certainty then the crowd's input could be leveraged, with greater focus on the identified experts in the crowd. On the other hand, if the model is fairly certain in its output, then it can work automatically, with no reliance on the crowd, which will be more efficient in both time and cost. Furthermore, a ML approach would allow for the MCS to scale much better (greater use of the data will be made if it can be trusted and a ML algorithm can help with this) which is one of its key requirements for MCS to be useful. However, these two research fields are quite large on their own and therefore joining them together will require plenty of additional work. Having a framework that simplifies the process of data collection and user participation, application of ML, and then mixing the two would be beneficial as it will allow the the work to be focused only on relevant parts without needing to "reinvent the wheel" for the remaining components of the system.

1.2 Project Goals

This project is motivated by the problems described in the previous section. It aims to facilitate or ease future work at the intersection of mobile crowdsourcing and machine learning through simplifying the workflow for researchers and anyone else who requires mobile crowdsourcing.

The project's main goal is to provide a software framework which alleviates, partially or completely, the need to design and build a new system for every new data collection or mobile crowdsourcing task, thus saving a lot of time and effort to interested parties. This goal creates several requirements for the framework:

- The framework should provide a way to specify in a high-level way what is to be done to perform a mobile crowdsourcing task. The specification process should not require any

technical knowledge and implementation effort, given that the framework users may be non-experts (for example, a journalist leveraging the crowd to collect photos from an event or a ML researcher with little mobile development experience needing mobile sensor data). To achieve this, the framework should offer a collection of common functionality units — plugins, which can be combined in the high-level task specification way to achieve the task's goals. If all the functionality is available, then there should be no need to do any development work whatsoever.

- If on the other hand, there are parts of the MCS task that cannot be solved with any of the available plugins, then the framework should make it most convenient and straightforward to extend its functionality, and for the newly-developed plugin to fluently interact with the rest of the system. This will allow the developer to reuse everything that's available in the system while only having to focus on the novel parts of their specific task needs.

- It is imperative for the software's generality and adoption to allow these novel plugins to be contributed to the collection of framework plugin solutions so that others can later make use of them, thus gradually growing the project's usefulness. It is necessary that the development and contribution of a framework plugin imposes a minimal technical burden to the developers, given their potentially different backgrounds. This requirement means that the framework must be oriented towards third-party developers — demanding no knowledge of the overall framework functionality but must only understand how to implement the plugin itself. This also requires that the system is designed for decentralized plugin development and distribution.

- The software must not impact negatively the user experience on the mobile client by abusing the device's limited resources or affecting the phone's performance and usability. This means that the framework should be developed in agreement with mobile development best practices. In addition, the impact of plugins needs to mitigated without restricting their functionality.

- Preserve privacy — the user's personal data is often the main target of a MCS task. However, such information can often be misused so it needs to be treated with care. Furthermore, given that third-party plugins are to be encouraged, additional privacy concerns arise. The framework should ensure the user is in full control and understanding of what data is being accessed and how it is being used. Such guarantee can be made only by the framework completely owning the user data and controlling the plugins' access to it.

To meet the main goal and the requirements derived from it, this project sets out to create a mobile application which acts as an extensible framework on the mobile device. The application will allow its users to browse MCS tasks — descriptions of work — and view their details and requirements. The application should enable the users to participate concurrently in multiple tasks, given that this does not violate the integrity of any of the tasks. The application is also to provide the necessary infrastructure and supporting services to enable the functioning of the framework plugins. It is the framework application that will automatically manage the entire task execution process by orchestrating the needed plugins to perform the required work. The mobile framework should be as little dependent on the cloud back-end for the execution orchestration as possible; this will allow for more robust execution, make the framework more scalable, and will enable certain location-based tasks that may have connectivity limitations as a side effect.

The plugins are only meant to serve as data generating units. They should be running on the device only when started by the framework. In any other case, the framework must ensure they do not run (and thus, impact the device's usability) because in such cases the plugins perform no useful work for a task which should be their only purpose. The plugins themselves are to be separate mobile applications. They are to be completely decoupled from the main application in terms of development and distribution — developed by third-party developers and provided to the user through the various mobile application distribution channels, or through the framework's cloud back-end. The framework must ensure that plugins are easy to develop and that only minimal development overhead is created for requiring them to work with the framework application. Lastly,

it is important that the framework application establishes total control over the plugins and provides this control to the user. It is to be used to manage the lifecycle of the plugins but more importantly — the access to the user's data. The framework should become the single point of trust for the user data; it should securely hold any data generated by a plugin and completely control the access to it. Through the framework, the user should be allowed to manage the plugins' access and thus, manage their own privacy.

In addition to the mobile framework client and any developed plugins which will be the main focus of the project, the objective is to also create the server back-end part of the framework in the form of a cloud service. It is meant to enable the high-level MCS task specification and plugin distribution process, described in the requirements. The server is also to provide all the necessary infrastructure needed for managing user participation in the tasks and the collection and use of the crowdsourced data. If needed, the server should also collaborate with the mobile client in the orchestration of the task execution. For example, if dynamic interaction between the cloud service, the task creators, and task participants is needed, then such client-server collaboration will be important. In addition, the current project aims to provide the design and lay the foundations for future work which is to enable the framework to perform more sophisticated execution of tasks such as interleaving machine learning and mobile crowdsourcing plugins in a task execution and combining their results.

Finally, given that it is in the very nature of MCS tasks to target the participation of many users with the data they generate, it is imperative that both the server and the mobile client are designed for performance and in a way which permits the overall system to operate at a large scale. For example, it is crucial that the client-server communication patterns are made to scale for many users, given the expected large amounts of data uploads, task sign ups and updates.

1.3 Contributions

The first contribution of this project is the presentation of the current MCS literature and the identification of the lack of research work which investigates the capabilities of combining mobile crowdsourcing with crowdsourced machine learning in one integrated system.

The main contribution is the presented Hive framework. It is a framework for mobile crowdsourcing that identifies and addresses the new challenges that mobile devices present in contrast to wireless sensing — putting the human in the loop of a task's workflow, having a strong focus on data privacy and device usability, and generality of task specification. Due to time constraints, the project does not address the mobile context and data reuse challenges but it proposes design and discusses how to proceed with their realization. The framework's novelty comes from addressing the above challenges while meeting its main goals — providing a straightforward way to declare MCS tasks and managing many of the common aspects of a MCS task. Hive has been evaluated to be more scalable and to have better mobile performance than the only other existing framework with similar goals — Medusa. Hive makes it possible to reuse existing functionality when specifying tasks and thus may require no implementation effort. What is unique about Hive is that its generality relies on the design choice of making the framework extensible by third-party developers which is a much more realistic and scalable approach than a centralized solution.

The last contribution of the project is the design and implementation of the mobile client which is part of the Hive framework, and more precisely, its main framework application-plugin application relationship. The presented work creates a mobile application that acts as a framework to others, is able to control their lifecycle and orchestrate them into execution to reach a larger goal. The framework has full control over sensitive data and mediates any plugin requests for it. Yet, the plugins are separate applications which are independently developed and distributed, the developers are not required to have any knowledge of how the framework's internal system functions, and have full functionality freedom. This design is not specific for the purposes of a MCS framework but can also be adapted to other systems that require similar mobile application framework with third-party plugin architecture. To the best of the author's knowledge, such work has not been presented for the present mobile smartphone operating systems. Such task is thought to be more difficult, given

their stricter application constraints and control, compared to general-purpose desktop operating systems where such systems are not uncommon.

1.4 Report Outline

The rest of the report is organized as follows. Chapter 2 describes the relevant literature. It focuses mainly on MCS solutions but also discusses relevant ML literature. The chapter also gives an overview of the technology used in the project. Chapter 3 describes the design principles that guided the project and presents the resulting architectures for the mobile client and server cloud service. Chapter 4 discusses the details of the implementation, what practical choices were made, the problems encountered during the implementation and how they were resolved. Chapter 5 critically evaluates the Hive framework and compares it with the few other solutions. Lastly, Chapter 6 provides an overview of what has resulted from the this project and presents pointers for future work.

Chapter 2

Background & Literature Review

The goals of this project set it within the context of mobile crowdsourcing (MCS). However, given the long-term goal to integrate MCS with machine learning, the ML literature, in particular obtaining truthful information from untrustworthy crowd, will be reviewed briefly. The following sections discuss the relevant literature for both and provide the necessary background.

It is important to point out that during the review of the relevant literature, it was observed that there was very little if any significant research at the intersection of these two topics. While MCS often involves analytics, these so far have come down to aggregate analytics. Using machine learning algorithms specialized for crowdsourcing will enrich the capabilities and robustness of such systems.

It is the main goal of this project to facilitate work in precisely this area by removing the need to build MCS systems for the purpose of training ML algorithms or working on crowdsourcing machine learning. In order for the project to meet its goals, the framework produced should be general enough to allow for different uses of MCS and interleave it with machine learning. Furthermore, the challenges typical for MCS and yet not problem-specific should be solved by the framework. Finally, it should be possible to implement the use cases found in the literature. All of these are discussed below.

2.1 Mobile Crowdsourcing and Sensing

In the most general case, crowdsourcing deals with problems that are too difficult for computers yet easy if not even trivial for people to solve. Hence, solutions are sought by assigning micro *tasks* to multiple people – *workers*, and relying on their efforts and expertise. Multiple workers are assigned the same task so that greater certainty about the answer is obtained. The issue of trust is discussed later on in this chapter.

Crowdsourcing has gained traction with the introduction of gamified techniques introduced by von Ahn in [31, 32] where people enjoy playing a game and the crowdsourced data is a by-product. Such techniques may be difficult to devise for some problems, however, and hence in these cases users would lack motivation to contribute.

On a larger scale, the creation of the Amazon Mechanical Turk (AMT) [20] marketplace has been quite beneficial to the crowdsourcing research community. AMT offers an infrastructure for creation of tasks – Human Intelligence Tasks (HITs) – which are then offered to a human workforce. Common HITs involve labeling images, translating snippets of text, and filling in surveys. In the relevant literature that was reviewed, there were only one or two cases where AMT had not been used as a part of the proposed solution.

2.1.1 Mobile Crowd

The use of smartphones enriches the possibilities in crowdsourcing — in addition to the crowd explicit input, the data obtained from the sensors can also be leveraged. This is called crowdsensing.

Lane et al. [16] give a good survey of the present mobile smartphone sensing capabilities, current systems, and key challenges. The authors distinguish the various scales at which mobile sensing is being researched. They are personal, group, and community (crowdsensing). Personal sensing works on an individual scale by obtaining data for one's own interest such as tracking exercise. Group sensing is similar to community sensing although at a smaller scale and usually the people involved know and trust each other which creates different privacy and ground truth characteristics, compared to community sensing. This project focuses on community sensing which is the least explored of the three due to the additional difficulties of scale and workflow, compared to the other two. Yet, experience in building personal and group sensing applications is relevant for crowdsensing ones as their workflow often is included in the crowdsensing one such as the data collection and possible preprocessing steps as is the case in [21].

Lane et al. [16] also discuss two sensing paradigms — opportunistic and participatory. Opportunistic sensing is a fully-automated approach to data collection — it relies solely on the multimodal sensor streams and perhaps local analytics. The benefit of this is that it lessens the requirements on the end-user who may have little incentive to actively participate. The authors note that this is especially true for community sensing scale where the immediate benefit from the application may not be lucrative enough. However, opportunistic sensing does not utilize resources well and can be susceptible to the context problem discussed in §2.3. Participatory sensing requires the user's active involvement — to initiate sensing, to approve and/or submit the data. It creates a bigger burden on the end-user but it leverages their intelligence as a workaround to the context problem. Lane et al. [16] expect that many applications of mobile crowdsensing will involve a hybrid approach. This view is shared by Ganti et al. [8] who suggest the term mobile crowdsensing should span a wide spectrum of user involvement. The current project has adopted this stance, with the addition of mobile crowdsourcing (i.e., provision of labels of any kind from the user). Mobile crowdsourcing can be thought to be at the very end of the participatory sensing spectrum. Hence, the abbreviation MCS is used — it can be read as both mobile crowdsourcing and mobile crowdsensing.

In their survey, Lane et al. [16] note that at present there is only "limited experience in building scalable sensing systems" and that the same is true for both participatory and opportunistic sensing applications. This observation has been made by several other researchers, too, such as Ganti et al. [8], Ra et al. [25], and Xiao et al. [36]. In their respective works, it is noted that acquiring a critical mass of engaged users (i.e., scaling up) is crucial to the success of any MCS application.

In their survey, Ganti et al. [8] focus entirely on mobile crowdsensing at crowd scale. They note that MCS has some unique characteristics. The authors make a strong case against building application silos — creating specific server and mobile client from the ground-up for each new application. They argue that it is difficult to design the applications and that it should not be the focus of the researchers to reinvent the wheel; furthermore, it's inefficient since different apps may handicap one another if they are not aware of their co-existence. For example, if they access the same sensors, the apps may cause erroneous data or more likely, one may starve the others. Hence it will be better for one application to orchestrate them because it will have a holistic view of what is going on. Furthermore, any processing the data undergoes, may be relevant to the other app and thus reusable.

Ganti et al. [8] suggest that in order to overcome the limitations of MCS application development and deployment, there should be a "unifying architecture" that allows developers to specify the application's data needs in a high level language and identify commonalities of the requirements among applications. The system should then identify which smartphones are applicable, and delegate with instructions to the phones.

2.1.2 Use cases

This section details the various applications of MCS that were found in the literature. Studying the use cases, the applications' architectures and considerations that have been made provides valuable information for the purposes of this project since part of the requirement is for the framework to allow such applications to be built. There has been a great variety of uses in the literature and only a few are considered here. Other uses are, for example, constructing a noise map, obtaining

real-time traffic conditions, noise levels or pot hole locations [23], measuring the quality of biking routes, and tracking bird patterns and sightings, to name a few.

As the survey will show, while at a high level all system architectures include a mobile client that does the sensing and a server component delegating the work, the systems often differ in their details. Lane et al. [16] observe that "it is not clear what architectural components should run on the phone and what should run on the cloud" and that where various components run depends on the application's considerations. Such are privacy, reducing communication cost, better responsiveness, and sensor fusion requirements [16].

VizWiz [3] is a talking application aimed to help blind people by answering their queries in nearly real-time. Bigham et al. [3] achieve timely interaction with AMT by recruiting the workers while the user is in the process of taking a photo of what interests him/her and recording a voice query. Both the picture and sound are then uploaded, with the sound being transcribed on the server. Then an AMT task is created through the AMT API. Nowadays, it will be possible to transcribe the audio locally, hence saving bandwidth and time. The authors create the quickTurkit module which has the goal of managing the workers. Because the goal is to allow blind users to have an interactive experience (i.e., allow more than one related questions), quickTurkit engages the workers with additional work so that they remain available until a new query arrives or until some timeout period. Worker replies and instructions are sent to the phone as text which then uses Text-to-Speech (TTS).

A use case that is described is finding the location of an object close to the user. The crowdsourced task is the question of how much the person should move and rotate. This information is then sent to the mobile phone which relies on the accelerometer and compass to find out the angular cosine distance from the direction in which the camera is pointed at to the object of interest. Then, the information is passed to a local sonification module that guides the user with sound of how much and in what direction he should move.

In their paper, in addition to the research contributions made, the authors provide good insight of the unique challenges blind people have and how beneficial a crowdsourcing tool such as VizWiz can be.

Legion [18] is a system that allows a real-time control of various existing interfaces by relying on the crowd's input. The system consists of a client that controls the interface and the user selects which parts of it are to be delegated to the crowd. Then the client streams a video of the UI to the server. The server is responsible to forwarding the stream to a web interface through which the crowd provides input. The server also has to mediate the crowd's input into one UI command, and recruit and manage the workers. The server uses quickTurkit [3] to maintain an active list of workers.

Lasecki et al. [18] offer different ways of unifying (mediating) the crowd input at the server. Because of the nature of the problem of controlling a UI(for example, trying to navigate a robot towards a given goal), it is beneficial to have effectively only a small subset of users controlling the interface at any small period of time. The evaluation confirms this, and in fact, the best cases are when leaders – individuals who at a given point have had the most support from others through action agreement – were given the explicit control. This is very similar to a basic crowdsourced machine learning algorithm where every participant's trustworthiness is evaluated. Hence the results hint at the usefulness of a machine-learning trust approach.

Lasecki also presents real-time conversational assistant system – Chorus – in [17]. It mediates crowd input and presents the crowd's replies as a single agent to the end user; the goal is to answer the user's queries. The paper is scarce on detail about the architecture but the author discusses a multi-tiered reward scheme that should keep the crowd engaged. Each worker is rewarded per interaction; if the worker agrees with the mediated response, then the reward is bigger. The reward is greatest for the worker who proposed the mediated response. Such incentives could again be useful in the crowdsourced machine learning where experts and non-experts are paid according to the value of their contribution.

SoundSense [21] is framework that models sound events and is able to recognize the context the smartphone is in. While it is meant to work on a personal, not crowd scale, it provides useful insights. It adapts to the usage patterns of the particular user through unsupervised learning — it identifies new contexts and then asks the user to confirm the new context, or they can command the application to no longer track this context. Because it deals with sound recordings, privacy can be quite an issue. Privacy is one of the reasons why SoundSense does all the processing locally and never uploads the sound recording. It actually does not have a back-end server.

Another interesting feature of SoundSense is that it performs a two-tier analytics. First, it applies the supervised learning algorithms that were deployed with the app to decide if the sound is voice, music, or neither (ambiance). In the second step, a specialied algorithm is applied, depending on the first step. In the case for ambiance, the algorithm is updated with the newly-discovered contexts. SoundSense has a pipeline architecture on the mobile phone. After obtaining the raw input, the application preprocesses it by segmenting it into frames. Some frames are filtered out before the remaining are presented to the first-tier algorithms.

Sherchan et al. [30] present a platform – CAROMM – which focuses on the contextual aspect of sensing and the data collection process in MCS. CAROMM's long-term purpose is to collect sensed data in real-time, aggregate and collate it, and correlate it with social networks such as Facebook and Twitter. Then, interested parties from the social circles can obtain real-time information through a service that runs analytics on the data streams and sends queries to the mobile devices. The aggregated information could be in the form of light, noise levels as well as multimedia and social network updates.

However, in contrast with this project's goals, CAROMM is not meant to execute MCS tasks of various types, nor accept their specification through describing executable steps the task is composed of. Instead, it is intended to provide an infrastructure level which can be leveraged by a single service making use of the collected data. For example, in the presented work, the authors build a localized social-oriented service on top of CAROMM that allows for running analytics on a per-location basis and also performing real-time queries about locations using fresh mobile data and social network updates. Furthermore, CAROMM's main goal is the continuous collection of data on the mobile device and its accurate reporting to the server with minimal performance impact. This is in contrast with the goal of this project to obtain data in steps, only as a means towards the accomplishment of a well-defined task. Lastly, the authors do not aim for CAROMM itself to be extensible (and certainly not by third-party developers) but instead its functionality is to be wrapped around by a service that runs analytics. For these three differences, even though the authors of the platform refer to it as a framework, in the context of this project's goals CAROMM cannot be thought of as a framework but as an infrastructure layer, a platform. The differences in goals compared to the current project mean that the design direction set for CAROMM will be significantly different from this project's. For example, there are very scarce considerations made about data privacy in CAROMM.

Nevertheless, the data collection insights the authors present are worth taking into account. The core of the work presented in [30] is the data collection model and system which is meant to run on the mobile client. The authors' primary goal is to improve the client performance according to the cost models they develop. They achieve this goal through a data mining approach. The client has five modules in total — a UI that controls the user preferences and initiates sensing, a module that manages the interplay between the other modules, a module that interfaces with the sensors and manages media, and a module responsible for communication with the cloud and uploading the data. The last module is the core one — it deals with data analysis. It incorporates local analytics to cluster the sensed data and only sends updates to the cloud if there are significant changes — a new cluster. The significance is a user-controlled setting which directly impacts the phone's performance. The authors rely on an open source toolkit for data mining — Open Mobile Mining [11] to perform clustering on the sensed data. OMM could have been useful for the current project, if the crowdsourcing requirements are for sparse updates. Still, it was not possible to find any documentation or source code and hence its capabilities (and potential usefulness)

are unknown. In their evaluation, Sherchan et al. [30] report significant savings in battery and bandwidth according to their developed battery cost models, because the updates happen only upon context change. The results present a good argument for performing analytics on the data before uploading it, and also show that data mining may not necessarily be application-specific.

The authors do not go into much detail about the server side of their implementation since their focus is the client. Still, they use Amazon's Elastic Compute Cloud (EC2) to host a web service which is queried through REST. The web service performs the aggregation per location. The sensor data is stored on a non-relational database – Amazon SimpleDB, while the media data from the user – on a relational database – Amazon Simple Storage Service (S3).

TurKit [19] is a toolkit that facilitates crowdsourcing but without considering any usage of mobile smartphones. It provides a library of function calls that interface with AMT, hiding away the entire process of crowdsourcing. TurKit expects that AMT tasks might take a lot of time to run and hence the program itself might crash. Hence, it opts to store the outcomes of remote operations (such as AMT Human Intelligence Tasks), while replaying the logged local operations since they are cheap and fast to run. This way, if TurKit is restarted, it can quickly restore to its pre-crash state. The authors provide the syntax constructs and their semantics for programming their "crash-and-rerun" model.

2.1.3 Existing MCS Frameworks

The literature review has identified only two mobile crowdsensing frameworks that share some of the motivation and goals of this project. The main difference with the current project's goals is either the lack of server execution support, missing support for third-party development and framework extensibility, or no straightforward ways for defining MCS tasks.

Medusa, presented by Ra et al. [25], is a framework comprised of both a mobile client (Android) and services running on a server infrastructure. Its goals and motivation come closest to the ones of this project.

The Medusa framework offers a high-level language for the specification of new tasks. The language allows the user to input the steps needed to complete a task and what actions should be done at each step — i.e., to define a state-action model. The task provider (referred to as *tasker* or *requestor*) can also specify what is to be done if a step fails thus enabling a more robust execution. Each step of the task specification corresponds to a type of an executable library which is developed and provided by the framework's core developers. The libraries are well-defined in their purpose and can be used in different tasks. This facilitates reusability and simplifies the task creation process — the tasker need not implement the steps if they are already available but only piece them together through the high-level task specification language. The centralized provision of the libraries enables the authors to maintain control over the kind of functionality the libraries possess which allows them to better preserve user privacy and the mobile device's usability. For example, before any collected data is uploaded from the phone, the user is asked for permission because it is part of the functionality of the only library with connectivity that is provided by the framework; any other library with Internet access is not allowed. Medusa supports stages with various levels of user involvement – from none (e.g, processing video) to a HIT such as recording a video or providing a label.

The framework comprises of services some of which run on the the smartphone, while others — on the cloud infrastructure. They are built with concurrent task execution, robustness and privacy in mind. On the cloud, the server accepts new task specifications, manages the running tasks, stores on a persistent storage the state of task progress per individual participant, and pushes new steps to be executed to clients, according to the task script. On the client, stages which are part of tasks the user has signed up for are executed, possibly in parallel, and status updates are sent to the server.

The Medusa workflow begins on the server where new tasks are specified in text – in XML – which is later verified according to the framework's requirements. Relying on XML means that

the tasker needs to understand its form and know what stages are available, and what all the parameters that each stage library requires are. A better way to do this would be to have a UI that hides away the XML and presents the available library stages, thus making it easier to specify tasks. Using such an UI is important, since some of the targeted groups of taskers are non-experts and are not expected to have technical knowledge, as Ra et al. [25] point out, too. A UI will also enforce the state and overall requirements to which the framework is committed, and hence simplifying the post-processing of the XML. A flexible UI with appropriate stage-specific constraints can be challenging to design, however. Once a new task is submitted, the core Medusa module – `Task Tracker`, is initialized with the task parameters. This module runs on the server and orchestrates the entire execution, receiving updates from both other server components and the mobile clients. The system creates one instance of the module per task. The module uses another part of the system, the `Worker Manager`, to recruit participants. The manager is told using the task specification how many workers to recruit and with what incentive. The `Worker Manager` relies on Amazon Mechanical Turk to publish and advertise tasks. Potential users are expected to visit AMT through their mobile devices, browse through tasks and sign up for the ones that interest them. The reliance on AMT has both advantages and disadvantages — from one hand, a community that is already used to crowdsourcing is leveraged, but on the other hand, using AMT limits who can participate by requiring that users have an AMT account. Also, the framework cannot fully control the task execution process but is instead dependent on AMT, and its availability and regulations. The alternative is to completely circumvent the usage of AMT through the provision of equivalent functionality in the framework. This would give the framework full control over the task browsing and sign up process, and provide the opportunity to present the tasks to the user in a way that is tailored to the MCS and ML tasks which are the sole purpose of the framework, unlike AMT which caters for a much broader set of possible tasks. Such approach can open possibilities for a more flexible interaction between the mobile clients and the server.

The `Task Tracker` gets notified by the `Worker Manager` whenever a participant sign up. The participation of different users for the same task can happen at different times which is why the framework makes sure to separately manage the execution for each individual. The `Task Tracker` is the module that orchestrates the execution for every mobile device. The module sends instructions through a notification service to the mobile device, telling it what stage it should execute and giving it any additional parameters it will need. The mobile device itself does not have the entire description of the task(s) it has signed up for. From a design perspective, it can be thought of as a execution unit that runs the binaries it is given. The mobile device only has to respond to status update queries from the `Task Tracker`, and, upon stage execution completion, notify the server about it, and provide it with its state information. The state includes pointers to the data produced. This state is stored in the `Task Tracker`'s table which contains the state information of all participants for the given task.

On the client side, the Medusa framework comprises of a stateless `Stage Tracker` which is responsible for handling the received notifications and downloading (and caching) stage executable libraries when they are needed. The main purpose of the `Stage Tracker`, however, is overseeing the execution of the stages and interfacing with the user. The stage execution itself happens in `MedusaBox`. It tightly controls the access of the binary to the phone's resources. The stateless stage tracker refers to the above-mentioned lack of knowledge of the task specification on the mobile client. This lack results in no knowledge of how far has the mobile client progressed in executing the task. The authors justify this design choice ('dumb smartphone design principle') with making the mobile device more robust — they have opted for storing all of the state of each task for each of the involved participants on the server; if the mobile device crashes, it is of no importance because the relevant state has been saved on the server and the client can be told from where to restart execution. It is not clear, however, why this would have been any different if the task and stage progress was stored on the phone's persistent storage, and the client simply resumed itself, making use of the Android built-in process management. This could have resulted in faster recoveries after a client crash with significantly less strain on the server. The server load is in terms of storage — as there is a need to track all the state of every participant in every task. The strain on the

server is also communication one — the device is completely dependent on the server and has to ping it every time it needs to do its next step. Furthermore, if tasks self-manage, this would have reduced the dependence of the client on the server, making possible the execution of tasks where connectivity is lacking which can be a common problem given the inherent mobility of smartphone devices.

Medusa relies on AMT for another purpose — notifications. The server leverages it for initiating new stage steps via notifications it sends. This is done by using AMT's API to send an SMS with state and stage parameters, specified in XML which is then parsed and interpreted by the client. It is not very clear why this step has been complicated with the use of AMT messaging; the authors report it to be quite slow, relative to any part of the overall system running time (excluding steps with human involvement). This delay impacts the applicability of the framework where close to real-time crowd interaction is necessary. The Android system used could have been utilized with its push channels.

Overall, the Medusa framework addresses several challenges that are present in MCS and provides a valuable insight of how the problems can be solved. The way tasks are to be created — through a composition of preexisting stages when available, and in a high-level language meant for non-experts, and the different levels of user involvement that the framework supports are characteristics that overlap with some of the goals of this project. Furthermore, multiple tasks can be executed at the same time, as the user is enabled to switch between interacting with the executing stages. A key difference, however, is that the availability of stage libraries depends on the framework owners. There is no support for third-party developers to contribute new libraries in a distributed fashion, and more importantly, there is no functionality in the framework that allows the developers to build the libraries as plugins without needing to learn how the entire framework functions. Furthermore, Medusa by design is entirely focused on tasks being executed on the mobile client and the server is to coordinate and orchestrate the execution; this can create a barrier for creating tasks that involve the training of a machine learning algorithm, or its usage in conjunction with a mobile stage. These differences have affected some of the decisions made during Medusa's design and implementation. They should be taken into account, together with the criticism of some of the techniques detailed, when working on the current project.

PRISM is a framework presented by Das et al. [5]. It relies on pushing untrusted application binaries to the mobile client (Windows Mobile 5) provided from task requestors — the taskers can use any library in their application which is then packaged as a binary and made available to the PRISM server to distribute. This allows for great generality because any binary that can run on the mobile client will be ran by the framework.

There are problems with such approach, however. The authors claim code reusability. However, it is not on the same level as desired by this project — while indeed one can make use of pre-existing libraries, when end-to-end functionality is considered, i.e. an executable task or a well-defined part of a task with a specific purpose, there is no support from the framework. There are no 'modules' offered to a tasker who can then chose to combine these to form a task. It is entirely up to the tasker to piece together on their own any libraries they need to form a working application. This means that the taskers still need to design and implement an entire application, building any control flow, infrastructure and persistence layers that the application may need. The benefit of using the framework in conjunction with the developed application is instead in the application's distribution to relevant participants. It is not in the task specification, orchestrated control flow, and provided additional services. Lastly, because the binaries are packaged per task, their size is expected to be big, and if there is more than one task using the same library in its binary, it will be duplicated. This is undesirable for a mobile device with constrained bandwidth.

Since the binaries are not certified, it is imperative to protect the phones from malicious applications. They are executed in a sandboxed environment using system call interposition. The sandbox blocks all system calls to the mobile OS and instead only exposes a controlled API. The sandbox is designed with additional measures relevant to mobile sensing. Such is resource metering – the client keeps track of per application and cumulative resource use and controls applications based on set

policies. The control could include slowing down the use of the resources, regulating the resource access time, and also killing the entire application. In addition, the mobile client allows the user to set sensor access preferences. Even with resource metering, a malicious application could still take advantage by slowly doing work in the background within the policy limits. Hence, the sandbox has the feature of forced amnesia. It is based on the assumption that most crowdsensing applications do not have long-running tasks; instead, after performing some sensing and preprocessing, they would upload the results. Hence, the authors propose wiping the application's state at a regular interval if it is still running. This is a strong assumption and may affect the generality of the framework. In addition, the mobile client allows the user to set sensor access preferences.

PRISM's authors argue against using a pull approach for distributing tasks because that could cause large server load. Instead, the authors opt for a push approach by having the clients register with the server and the server would push only relevant tasks. It is argued that this achieves better scalability at the cost of some privacy (because of the registration). The authors argue that the users will poll all the tasks but participate in only few which will lead to large strain on the server and unnecessary data sends. This signifies the way tasks are thought of — only as their executable binaries. If, instead, tasks were considered as a meta description together with the binary that belongs to it, than a user pulling the task descriptions would not cause much load, given the often relevantly small sizes of meta descriptions.

Registrations time out and in follow-up re-registrations the same user cannot be recognized because of an added anonymization layer. However, this extra layer also would prevent the use of crowdsourced machine learning since a model of a person's reliability and expertise is built. Hence, such approach is not applicable, at least not without further modifications. The concerns regarding the pull method could have perhaps been addressed with a better query design. Both should be considered with this project's work.

PRISM relies on a sandbox (through system call interposition) to control all the binaries. However, it is thought not possible for this to be done in Android (the OS that was chosen for this project) by a standard application, as briefly described in [28]. This is because of the unique start up process the system has for applications — only a special process, `Zygote`, has the privileges to start new processes. No application can do this. The process contains the Dalvik VM and all the core libraries linked. `Zygote` forks itself for every application. Thus, each application is in a separate Dalvik VM which also acts as a sandbox. None of the application processes is able to perform direct system calls, instead it can only use the Dalvik VM. However, none of the application processes created by `Zygote` is able to monitor the system calls of the Dalvik VM and therefore a standard application cannot create a sandbox environment around another application.

With the applications from the previous section and frameworks discussed here, a few observations can be made. Some techniques are often useful across different use cases. Preprocessing the sensed data can resolve some privacy concerns. It is likely to be application-specific, however, so means to do this should be provided in this project's mobile client. In addition, removing outlier data points (which could be due to poor sensing conditions) could also be beneficial. One step further could be the application of data mining techniques locally on the phone before sending the output to the server for aggregate processing and machine learning, such as in [23, 21]. All of these can save bandwidth and upload time. Furthermore, there will be less strain on the server which can be a factor when at a large-scale participation, especially if there are tight time constraints.

2.2 Crowdsourcing Machine Learning Approaches

The topic of obtaining ground truth from many sources has been well studied in the machine learning community. With the advent of crowdsourcing facilities, mainly the Amazon Mechanical Turk, the research has accelerated and scaled up. The ML techniques are used on data obtained from many users(*workers*) and the aim is to infer some target label about every data point. Unlike normal supervised learning, there is no ground truth to be used during the training phase. Instead, the algorithms use the ones provided from the crowd.

The involvement of people to provide labels creates additional problems, however. The main one is accessing the trustworthiness of each participant. Usually, no assumptions can be made about them and hence they cannot be trusted individually — a participant might be lacking the knowledge to provide the correct answer, may have misunderstood the task, or could be acting in an interest that conflicts with providing the right answer. It is therefore necessary that a large number of workers is involved so that any 'noise' in the provided solutions is drowned. However, this can be expensive and time-consuming and therefore it is essential that trustworthy participants are queried more often while others' input is given less individual significance.

In the mobile crowdsensing context, trustworthiness is also a function of phone quality — the sensors can be poorly calibrated or misbehave in some context (such as indoor GPS), as Ganti et al. [8] note.

The algorithms that have been studied all have the common feature that use graphical models to represent the relationship between the data, workers, provided labels, and ground truth. In addition, some authors also model the difficulty of the task [2, 34]. The data and the provided labels are the only observable variables while the rest are latent and need to be estimated. The estimation is done through the computation of a Maximum Likelihood or, if a prior distribution on the latent variables is introduced, Maximum A-Posteriori estimation. The way this is done in the reviewed literature [2, 26, 27, 33, 34] is through an iterative approach, using the expectation maximization (EM) algorithm. When some ground truth can be established for some of the data points, it can have a positive impact on the performance, as Bachrach et al. [2] show. This can be relevant to the current project. For example, by verifying some of the users' input with sensor data, if there is strong certainty about the sensor being correct.

2.3 Key Challenges

In this section, some of the key challenges that any framework in the context of MCS needs to consider are discussed. set out in the Goals section (§1.2). These are mostly inherent from the challenges specific for MCS (and not present in the subject of wireless sensing, for example), yet the addition of interleaving machine learning algorithms into the workflow will cause some requirements on the format of the data.

2.3.1 Mobile Context

The mobile context arises from the smartphone's inherent mobility — the phone could be in circumstances not expected by the task provider and hence the data obtained may be incorrect. Mobile context has been identified several times in the reviewed literature [16], [8], [15] as an important issue that needs to be addressed in MCS applications. The framework should aim to enable precise context specification as part of the task declaration which is then verified at the client before and during the execution of a step in the workflow.

One way is to rely on participatory sensing, i.e. on the user's involvement, to fix it. However, this will often be either too frustrating for them, or incorrect, or both. There needs to be a mixture. The reviewed techniques for local analytics are a viable candidate for at least reducing the problem. Furthermore, full use of the underlying operating system's APIs should be made since these APIs are likely to be optimized and relying on internal functionality and knowledge not available for application developers. For example. recently geofencing and simple user activity recognition have been introduced to the Android system.

2.3.2 Privacy

The user should be given enough control of what the application will sense and upload to the server. This is especially true for identifying information such as sound, location, photos and video. As it was seen in the literature review, a potential way to minimize privacy concern is processing the raw data into a feature vector or similar representation that does not identify the user but is yet useful to the task provider. Medusa [25] always confirms with the user before performing any upload.

Ganti et al. [8] suggest data perturbation as an effective way to preserve anonymity – noise that is centered around 0 is added to every individual data point and so the aggregate is not changed but the individual information is no longer identifying. This can be only useful for sensor data, however, not user media and other input.

2.3.3 Phone Usability

The performance of the phone should not be compromised by running the MCS application. Otherwise, the users will not adopt the application no matter its merits. Hence, the entire client should act optimally on every step – collecting, processing and uploading data, interacting with the user, running analytics and detecting context. Data should be re-used where possible, and it should be uploaded only once if needed for multiple purposes. If there are multiple tasks active at the same time they may not only interfere with one another but also have a compounded effect on the phone's performance and usability (for example, creating a deadlock between two tasks which results in the user being unable to have their voice heard during calls). Therefore, there should be one application that has 'global' knowledge and is able to oversee what tasks the user is participating in.

2.3.4 Generality

The software should allow task providers to express their data needs and give the context in which they want the data to be obtained. This can happen either through a high-level specification like in Medusa [25] or through a tightly controlled UI that will also ensure correct specification and will not require verification. Either way, the specification of the task should be flexible enough to accommodate the various MCS applications found in the literature. Furthermore, the software should be general — it should provide functionality that is thought to be common among MCS applications and hence could be implemented only once, removing the need for the task creators to handle the design and implementation challenges.

2.3.5 Scale

Xiao et al. [36] note that scale is "key to the success of crowdsensing applications" because users are generally unreliable and hence the usefulness of the system suffers unless it has large user base. The authors discover that the research literature has rarely used more than 1000 participants [36]. Xiao et al. [36] identifies three issues for scaling - heterogeneity, burden on users, and network bandwidth demands. They are "ramifications of the deployment model in vogue today, where participation in each crowdsensing activity requires a separate application". In order to scale, the client-server communication patterns and the separation of concerns between the two will be crucial for scalability.

2.3.6 Data reuse

Data reuse can be a large boost, both to the participants in MCS but also the task providers because it will decrease the effort needed to provide and obtain data. It would be very beneficial if the system is able to recognize when the data the worker has already obtained or submitted is applicable to other tasks. Hence, both the phone and the server portions should maintain persistent storage and the context of the data. This will speed up the entire process. The data should be reused only if the client signs up to the task.

This realization gives another argument for the creation of a single application that manages all assigned tasks.

2.4 Technology Used

2.4.1 Android

The Android [14] mobile operating system is a system that includes a modified version of the Linux kernel, a set of core C/C++ libraries such as support for SQLite and media playback, the application framework, and the Dalvik Virtual Machine. The VM holds all the core Java libraries that an application might leverage and any Android-specific classes [22]. The system runs every mobile application in a separate instance of the VM, and hence each process is isolated from the rest. All of the application's parts execute in the same process (unless otherwise specified — but still in the same VM), and on the same thread — the main thread, unless the application does not create new threads, directly or through using particular functionality. The main thread is the only one where UI interaction can happen. Conversely, slowing down the main thread will result in unresponsive UI.

2.4.1.1 Terminology

An Android application is comprised of standard Java classes in combination with Android-specific ones that come with the Android SDK. The latter make up the primary parts of an Android application while the former are used only for supporting roles. An application is comprised of four major components — Activities, Services, Content Providers, and Broadcast Receivers, each of which has different usage patterns.

An Activity is a visual component through which the user interacts with the application. It corresponds to a single screen. It should only contain the view representation and possibly a simple controller of the underlying data that is shown.

A Service does not have a visual component but instead is meant to run in the background. It is suitable for long-running operations that should continue working even after the Activities that may have started it have been recycled by the runtime. The fact that it can live on its own sets it apart from a Java Thread which lives only as long as its parent Android component.

A Broadcast Receiver responds to system-wide announcements. These can be about a sensor or service being switched on or off. The broadcasts can also originate from other applications. A Broadcast Receiver is meant to be a lightweight component which is only created to handle an announcement to which it has subscribed. The handling itself should be very fast; usually the receiver will just decide *how* to handle it but not do it itself. Instead, the receiver will start the needed service to handle the broadcast or create a notification for the user.

2.4.1.2 Data Storage & Sharing

The last component of the four main ones that comprise an application is the Content Provider. It is one of the different ways to access Android's storage. For simple operations, one can directly use the file storage which each application is allowed by the system. By default, this storage is private to the application unless it is on the external memory card where access permissions cannot be enforced. Android also offers a working SQLite database which can be directly accessed through SQL queries. However, while it is convenient to do this for small applications, for more complex ones it is recommended [14] to use Content Providers despite that they are more complicated to build. Furthermore, Providers are needed if data is to be shared between application in an efficient manner.

Content Providers declare a unique URI path to which the particular Provider will respond — since a mobile device can have multiple applications with multiple Providers, it is important that each one of them defines a unique path; usually java package naming conventions are used. A Content Provider offers an API (and is the one that must be implemented) that behaves like a relational database. However, its underlying storage need not be a database at all; it can be the filesystem or even the Internet. It is up to the developer how they will handle the implementation. The counterpart to a Content Provider is a Content Resolver. A Resolver is what is used to access the data held by the Provider. The Resolver uses a URI to ask the Android system to access the

data corresponding to the given URI. The system will check whether the application using the Provider has permission to access it and will then instantiate the Provider which will in turn access the underlying data source and respond.

The benefit of content providers is that they are well supported by functionality in the Android SDK that deals with data loading on the UI. Content Providers are a key to making a complex UI that is robust and fast. The other benefit is that Providers can operate with URIs and a single URI can be made to address a collection of data as finely or crudely as needed — it is up to the developer. This is beneficial because URIs can be used to grant temporal access permissions to other applications to data that would normally be out of their reach.

2.4.1.3 Process Management & Lifecycle

A primary difference compared to software applications developed for computers is that Android applications do not manage themselves their own lifecycle. Instead, it is the operating system which does this. The process and memory management is done by the kernel which will kill processes with low priority. The priority is determined by the kind of applications hosted in the process (usually one). In turn, the priority of an application is determined by the highest priority of its components. Components are given different priorities depending on how important they are to the user — for example, interacting with a visible Activity is important while an Activity from another application that has been in the background for a while is unlikely to be needed; it is quite possible that the Android runtime will kill the second one, if it is the only component in the application. Furthermore, a Service performing work for a user is more important than one which is waiting for something to happen. In addition, the users actions affect the lifecycle of the components, too. For example, pressing the Back button on a device will destroy the current Activity.

The combined effect of the user's actions and the system mean that from a development point of view, the application cannot expect to have object references between its components. Instead, the components need to be designed to be independent and only work together through messaging (Intents). Alternatively, when it is deemed good design, one could use a globally shared singletons that provide the needed objects. However, this can have large resource implications and needs to be justified.

To adjust to Android's control over the lifecycle, the application developer has access to callback functions that provide different call guarantees and semantics — for example, `onPause()` is called every time an Activity goes into the background even if only partially. However, `onStop` which is called when fully in the background is not guaranteed to be called before an application is killed. According to Meier [22], managing the components' lifecycles is vital for proper resource management and good user experience.

2.4.2 Web services

A web service is a way of making functionality of a system available over a network to other other parts of the system or other systems.

There are two different approaches to building web services. One way is using a combination of Web Services Description Language (WSDL) and the SOAP protocol; this approach will be referred to as *WS-**. The other way to create web services is to follow the Representational state transfer (REST)[7] architectural design style.

The first approach focuses on the semantics of each of the components and their complex interaction between one another. The interaction is suitable if intermediaries are involved and it can comprise of several steps (i.e., stateful communication). WS-* web services rely on each component knowing precisely how to interact with others and the interaction happening through messages following precise formal contracts. In this way, WS-* web services behave just like components in a distributed system — the focus is on the interaction via the exposed API functionality of each component.

On the other hand, a RESTful approach focuses on the data in the system. There is a constraint on the connector semantics between components/services because REST services are most often

based on the HTTP standard which permits only the GET, PUT, DELETE, and POST operations. RESTful web services are centered around the representation and exchange of the state of the data (a Resource). RESTful communication is stateless and most suitable for a point-to-point interaction.

Chapter 3

Design

This chapter presents the principles that drove the system design and led to the software's architecture, also described in the chapter. The principles were chosen in order to meet the framework's goals outlined in §1.2 and address challenges discussed in §2.3. The main goal of the project is to provide a flexible solution that allows the fulfillment of various user-defined mobile crowdsourcing tasks and alleviates unnecessary development burdens. In the remainder of this report, a `Task` is defined to be a collection of concrete executable units of work which, if followed, achieves the desired outcome. Each such unit of work may require specific circumstances under which it is executed and will be referred to as a `Stage`.

3.1 Design Principles

3.1.1 Make A Framework

As identified in the literature review, building a software solution for every new MCS problem does not scale for the expected 'crowd' size in crowdsourcing and poses numerous challenges that can significantly slow down work in the area. Instead, there is a need for a shared functionality that offers solutions to common problems – whether these issues are to do with infrastructure on the mobile clients and servers, or the actual data collection and processing on either. A software library would be an improvement over an ad-hoc solution, yet it is still unsuitable as it will require that its user learns how the library functions before using it. Furthermore, the user still needs to do the entire work of building a system that integrates such a library and *controls* the task execution. Also, the user will have to implement any additional processing work that is not provided by the library, even if others may have already implemented such functionality for their own needs. These observations mean that a software library does not solve the problems motivating this project. A software framework, on the other hand, is intended to offer generic solutions that can be adjusted, if needed, to the needs of the user through modifying only its peripheral functionality. It also allows the provision of novel solutions aimed at new use cases and their future reuse in the framework. A key benefit of a framework is that it hides away the complexity of its functionality at a system level, unlike a library which does this only on a module level requiring the user to learn the intricacies of how the modules should be combined.

The core principles on which a framework is built are modularity, extensibility, and inversion of control. The software needs to be modular because parts of the software may need to be replaced by alternative ones in order to create a satisfactory solution for the task at hand. The richer the library of interchangeable modules, the wider applicability of the framework. Conversely, to increase its applicability, the framework needs to be extensible. During the initial design, it is impossible to predict all future use cases that may arise, and what their common and unique characteristics are. Therefore, the framework will certainly need to be modified or have additional functionality added. It is important that the design flexibly accommodates such extensions.

Such flexibility becomes possible only if each of the framework's components has a single, well-defined purpose and any interaction with it happens only through precise interfaces. On the server

side, as it will be seen, this is achieved through web services, while on the client – through the design of a interprocess interface.

The last important aspect when building a framework is maintaining inversion of control — it is the framework, not the user relying on it, that controls the execution flow through orchestrating the system's components. This makes possible maintaining the framework's core behaviour intact and keeps the complexity of its inner workings hidden to its users. It also allows to impose control over what the additionally provided extensions are capable of. Inversion of control is crucial to meeting the project's goals.

Collectively, these tree aspects guide a design in which there is a distinction between the core framework functionality that controls the behaviour of the entire system, and the peripheral points of the system that are to be interchanged, extended or modified. This distinction is sometimes made using the terms *frozen spots* and *hot spots* [24]. The frozen spots, as the name suggests, do not change often, if at all. On the other hand, hot spots are meant to be points in the system where module implementations can be interchanged, and new functionality added. The way this is achieved is through conforming to an interface required by the framework. In exchange, the hot spots often have access to an interface exposed by the framework which offers relevant framework core functionality.

3.1.2 Decentralized Extensibility

A principle followed in this project is designing with third-party developers in mind. The success of the framework depends on its applicability to problems and hence, extensibility. However, it is unfeasible to satisfy and even predict every use case. Instead, when there are new needs that cannot be solved by the current framework functionality, it should be made straightforward that a third-party developer can add just the new functionality in the form of a plugin – a new kind of *stage*. Such new processing stages should fit into the framework as if they are part of the originally provided functionality, yet it should *not* be needed for the third-party plugins to be packaged with the original framework as this creates a bottleneck on the framework's side. Instead, the plugins are to be distributed separately and on demand, thus making the framework's extensibility decentralized.

Decentralized extensibility is well supported by following the framework design characteristics described in the previous section as the framework's hot spots are ideal for plugin addition. A framework is already meant to have extensible architecture. Furthermore, the inversion of control means that the provider of new functionality does not need to know how to operate the entire system as long as the new plugin fulfills the required interface requirements. This is especially important since the framework developers can be of different backgrounds such as machine learning researchers or smartphone developers and hence may find it difficult to understand parts of how the core functionality works. To make it more suitable for third-party developers, however, the software should also actively guide the developers with what is required of them, and what is available to them. This can be achieved through self-documenting interfaces and precise error messages.

Some of the requirements the framework has to impose on the plugins are because of the decentralized principle. They are to ensure that plugin behaviour does not act against the framework's and users' interests. These requirements are derived from the principle discussed next.

3.1.3 Put The User In Control

The reviewed literature has shown that one of the key challenges faced by MCS work is letting the user decide what data is shared and respecting their privacy. For an ad-hoc application, such design can easily be achieved since the targeted user group is well-defined and usually the data needs are well-known in advance. Hence, the details of data privacy can be precisely agreed on. However, for a general framework that is to be extended, this is more challenging. Even more so with the goal of allowing third-party developers provide their own new processing work units — such plugins need to be transparent in their intentions. Transparency can be achieved by imposing requirements on the plugins through interfaces. That way, in order to utilize the framework, the plugin will need to

provide all the relevant information which is then to be presented to the end user. Furthermore, the system should be able to exert strong control over the plugin execution. This control can then be offered suitably to the end user to meet their own personal preferences. Such control can be achieved through relying on inversion of control and also limiting the capabilities of the APIs offered to the plugin — for example, any personal data access from a plugin should happen through the core framework functionality. The data is to be owned by the framework itself, and the plugin should ask for permission to access it.

Also, a much better privacy control can be achieved if each plugin has a single very well-defined purpose, and a complex *task* behaviour is constructed only through the composition of such plugin *stages*. Such approach not only facilitates good software development practices but also improves end user understanding because smartphone operating systems such as Android provide ways (through application permissions declarations) that hint at what an application will do internally. If a for example, a stage plugin that is meant to be only post-processing locally some sensitive private information but the system has reported it to be also using the Internet connectivity, then perhaps the user will not trust it.

3.1.4 The Smartphone Is Smart

A design strategy followed by the Medusa framework discussed in the literature review is giving the least amount of meta information to the mobile client, having a stateless client, and maintaining the client's progress on the server. The server is the one which orchestrates the workflow. However, this has significant negative implications on the overall system scalability. Such model puts a large communication burden on the server because after each stage completion or crash, the mobile client will need to update the server and ask for further steps to perform. This also creates a storage and computational burden — the server needs to maintain the progress of every smartphone and has to make any additional computational decisions for what the clients should do next. Finally, there is a usability issue — the mobile client is dependent on the server for further work. This means that no progress can be made while the client device is offline. This will slow down task progress, and even limit the applicability of certain tasks (for example, taking a set of photos along a route in the mountain where there might be no cellphone coverage). Given that limited connectivity is an important characteristic of mobile devices, such server dependencies need to be brought to a minimum. All of these issues will fast become significant problems if the system scales in number of users which is its main goal, considering the purpose of obtaining data at crowd scale. The solution to the above three issues proposed in this project is to take the load off the server — trust that the mobile client is capable of complex work, build execution self-management in it, and let it communicate with the server only when really necessary.

3.1.5 Performance & Scalability

The performance of the mobile device is one of the key challenges identified in the previous chapter. Given the decentralized extensibility principle discussed earlier, the framework must ensure plugins do not have negative impact on the user experience. Hence, the framework must put tight control of the lifecycle of the plugins, and the way the interaction between the main application and the plugins is performed. The stage plugins should run only as long as necessary and not any longer. Furthermore, communication between the client and server, and between the framework and plugins should be minimal in size and quantity, yet still giving control to the main application over the plugins, and also allowing the plugin to provide accurate and up-to-date information of its work.

Performance and scalability should be kept in mind when building the server, too. The main requirement is to be able to cope with large number of users and to handle data between web services in an efficient manner.

3.2 Data Model & Storage

At the center of the framework's goals is the collection and processing of data as a fulfillment to various user-specified tasks. This section discusses the modeling of the different types of data and the design decision of how to store it given the way it will be used. This section discusses both the client and the server aspects of the data model.

The data can be logically separated into two – meta information and the actual data. The meta information is modeled as tasks. A `Task` holds any information describing the overall unit of work. It is the highest entity in the hierarchy of the data model. It holds a collection of `Stages`, and each stage encapsulates any needed parameters. The other type of data is the actual data which is produced or consumed by a `Stage`, whether it is on the client or server. It is this actual data that is of interest to the user specifying the `Task` (the tasker). The above distinction is needed as the two kinds have different characteristics which affect their modeling and storage.

The data is produced by task participants on their mobile clients and then some representation of it is sent to the server. However, the client devices will need the meta information in order to produce the data. In Ra et al. [25], the Medusa framework relies only on sending the individual stage details because the authors follow a 'dumb smartphone' design principle, as described in §2.1.3. In contrast, this project's design decision of having smartphone clients self-manage themselves whenever possible means that the clients will require additional information that describes the entire task. This results in even larger overlap between the kind of meta information stored on the server and on the client. However, there are still differences between what is stored on one and what – on the other. It is important that the differences are identified for a few reasons. First, sending unnecessary data to the client only increases the communication load and storage requirements. While for today's standard, a couple of extra kilobytes are insignificant, if there is a very large number of tasks on the server (as is the case with Amazon Mechanical Turk) and the client queries them all, then wasteful data sending and storing can quickly become noticeable, both for the client and server. But even more importantly, the tasker may not wish to expose some information to the task participants. For example, he may wish to hide what kind of analytics (which will be a stage in the task execution flow, to be run on the server) are performed on the collected data, or who else is participating in the task, or what kind of trust machine learning algorithms are used.

With these considerations in place, the meta information has been modeled as the class hierarchy shown on Figure 3.1 where part of the model is shared between the client and the server, and while the remainder is not.

Note that most of the attributes defined on that diagram, such as `name, description` and flags relating to the status of the task or stage – `finished` and `successful`, are attributes that deal mostly with providing the end user – the participant – with feedback information. Such attributes can either be set as constants or set programatically by the concrete plugin implementations. The only crutial parameters in the data model are the `intentString` and the `parameters`. The `intentString` is named as such because of an implementation detail to do with Android. This attribute's value is the name of the unique executable work unit of which the `Stage` is the meta description of. The `parameters` attribute contains all the input parameters required by the work unit (plugin) and provided by the *tasker* (not the plugin developer). The parameters are interpreted by the plugin in its own manner. They can be completely different between plugin types. It is therefore not possible to design in advance for all possible parameters without also limiting future plugins. Instead, it is better to let the plugin declare what parameters it needs and these be handled in a general way by the framework. This is discussed in more detail in the server section (§3.6). Lastly, there are attributes that provide information about the input data the stage requires. If these are non-standard, then they could be added to the parameters Map and handled by the plugin.

The data produced by plugin stages is to be stored in files as a binary stream. This is suitable because of the data's heterogeneity and because it will be needed in its entirety, unlike the meta information which is to be accessed regularly, and often only parts of it queried or modified. This is the case for the produced data for both the server and the client. The meta information, on the

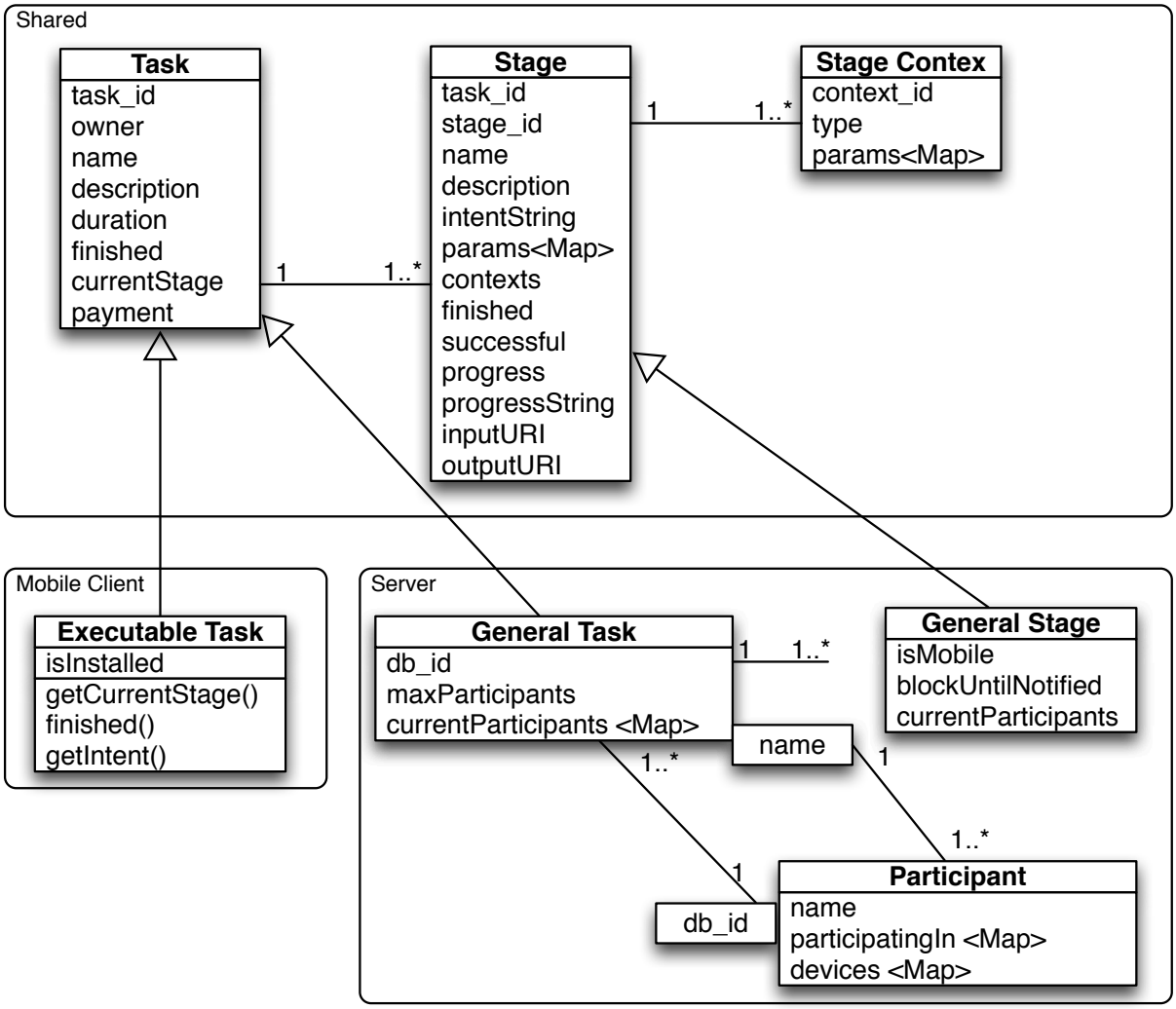

Figure 3.1: Task Data Model

other hand, is structured and only textual which makes it suitable for storing in a database. As seen in Figure 3.1, there is a chained has-a relationship between the classes in the order of Task, Stage, Parameters and Context. As it will become clear from the later sections in this chapter, the server uses the meta information only as a logically grouped unit – grouped per task. Such usage is referred to as aggregate, and the data – aggregate data [29] . In such usage pattern, a NoSQL database becomes a viable option as it offers horizontal server scalability. It is also allows for easier and speedier development because it simplifies the data storage and access model, compared to a relational database where the data would need to be normalized and spread across several tables, after which `join` operations performed on every query that needs the aggregate form — the entire `Task`. Since this project's focus has been predominantly on the client, there is no complex functionality that would need the advantages of a relational database (column-wise operations and rich queries). With the addition of such requirements, a relational database will gradually become a better option.

On the client however, the access patterns of this same meta information are different. Because of the mobile devices' small form-factor, there is limited space to display information. This means that that complex information such as a Task object is displayed over several screens. Because of the lifecycle of Android Activities (they can be disposed when no longer visible), and the design goal of not impacting the smartphone's usability, it is not suitable to share an entire object between activities, or to hold such objects in memory only to access a particular single field. Furthermore, some of the fields in the meta information will change on the client and the UI will need to

immediately reflect this change. If separation of concerns and low coupling are to be maintained, then the modifying components (which can be more than one) should not update the UI. Instead, the UI should subscribe directly to the model, and the subscription should be only for the bits of meta information that the UI displays. Lastly, the client relies on rich queries to operate — for example, only activate tasks that the user is registered for, or get the outputs of particular stages. With these considerations in place, a relational database is definitely better suited to store the meta information on the client.

3.3 Task Workflow

The `Task` has a collection of `Stage` work units. More precisely, the collection is designed as an ordered list. This forms a sequential workflow for the tasks — a task is complete once all the stages in the list have been executed in the predetermined order. This approach is in line with the workflow design of the Medusa [25] framework which defines connectors between stages to have only one output end. Allowing only for sequential workflow does impose restrictions on the task flexibility. However, there are a couple of factors that ameliorate this problem. First, each of the executable `Stages` can be a complex composition of services and interactions with the user, all working towards the common goal of the plugin stage. A stage could incorporate flows such as looping (iterative progress updates through iterative interactions with the user) and concurrent processing. For example, recording multiple sensors at the same time, or asking the user to provide labels for several pictures in a row with other activities interleaved. Such composition is best managed by the plugin itself. All the framework execution should do in these cases is to maintain the stage progress when the plugin lifecycle has been (temporarily) ended, so that it can later be resumed from the appropriate point, to control the plugin, and to provide the necessary functionality. Furthermore, as it can be seen in the data model in Figure 3.1, each `Stage` can contain pointers to the data it needs from other stages. In other words, the data flow is not sequentially pipelined but instead a `Stage` can access (if permitted) any of the data items produced by earlier stages, similar to the Medusa framework [25]. Since the framework mainly deals with producing and processing data, it is this data flow flexibility that is more important. Accessing the output of any stage allows for extra flexibility. For example, optional stages that post-process data while the original data is still intact and preserved; this means that later it can be used by some other stage and task which can improve data reusability between tasks. A distinction with Medusa is that any stage can produce any number of outputs, rather than only one, or a collection of the same-type variables. This enables a finer-grained possibilities for the stage plugins which can selectively get only some of the data output – for example, filtering by MIME type because only some of the types are supported by the plugin.

Together, the sequential control flow execution and the unidirectional data access provide simplicity while still a good amount of flexibility as it will be shown for the plugins discussed in the Evaluation Chapter §5. The simplicity is important from the point of view of the participant who is to understand what the task requires to be done before signing up, and it is also useful from a performance perspective given the limited mobile device resources — developing and running a complex execution flow engine may not be appropriate.

Despite the above points, while the design decision is a viable one for the mobile clients which are the main focus of this project, there can still be limitations for the kind of tasks that can be ran on the server, and these could be avoided with a more flexible control flow. This issue could be addressed by future work.

3.4 Framework Usage

Before going into the detailed design of the client and server, this section briefly describes the envisioned usage of the overall system. The specifics of each of the steps involved will be described in the appropriate sections.

The framework will offer a collection of already available plugins, able to solve common subtasks. One way to use it is to contribute a new plugin to that collection by registering its required and optional parameters for the new type of stage. The stage can be listed as part of the available framework solutions that can be combined to specify a task.

Another, and the more common starting point of using the framework, will be the desire to perform a new MCS task. The tasker will navigate to the task creation client provided by the server, and define a task with parameters such as what stages to include, what is their order and what parameter values they should be given, and how many participants should the system allow. The new task will then be available to be viewed by anyone who has the mobile client application installed and anyone can sign up to participate. If they do not have the needed stages installed, the users will first need to do this which can be managed by the mobile client. The mobile client will manage the task execution workflow; some steps are likely to generate data, while others — process it or upload it. On the server, the management of the participants and data will happen. Steps of the workflow that are on the server will also be executed. Ideally, the tasker will be able to monitor to task progress.

3.5 Client

The client plays a significant role in the project since most executable stages will be run on the mobile device. It is therefore essential that the principles discussed in the previous section be followed. The mobile client is a smartphone application. Its main purpose is to provide means for the user to participate in tasks and control their execution on the device, to show users relevant information (such as progress and stage description), and to orchestrate third-party plugin applications that are needed to accomplish a task. The orchestration is necessary to avoid data incorrectness, interference with the phone's usability, and for better performance.

The main application holds the core framework functionality which comprises of services and infrastructure built on top of the smartphone's operating system. These provide solutions to both the main application and, through it, to the plugin stage applications. A key decision to the overall framework design and implementation is to keep all the data generated by any of the plugins in the main framework application and completely control the access to it. This is done for two reasons. The first is derived from the principle of putting the user in control and respecting their privacy. Given that the framework encourages third-party plugins, it is not possible to control what the plugins do with the user data without restricting their functionality freedom (as does Medusa, which does not allow plugins using the device's Internet connectivity). Nevertheless, the user's privacy is of primary concern and there needs to be a way to monitor what is using the user data and when but more importantly, allow or deny for the access to happen. Such monitored and controlled access is only possible if the plugins can access the data only through the framework. By default, no plugin should have access to any data, even to the one output by itself at some earlier point for an earlier task. Permission to data should be granted only temporally and only for a most-precise set of data. The power to grant such permission should be held only by the main framework application, and through it — by the end user.

The second reason for this decision is to make the framework more flexible and efficient in the handling of data between plugins. Given that data is at the center of each task's purpose, such handling is bound to always happen. If the data is held by and accessed through the framework, then the access pattern becomes star-like with the framework at its center and the plugins at the edges. This will make the data access standard and will allow for access to data (if permitted) between any two stages.

The main application consists of several layers conceptually built on top of one another. The overall architecture is presented in Figure 3.2. Modules that are part of higher layers are designed to depend on the ones in lower layers. It is expected that modules in high layers are to change often or that new modules are added. Low layers are more foundational and their modules are unlikely to change.

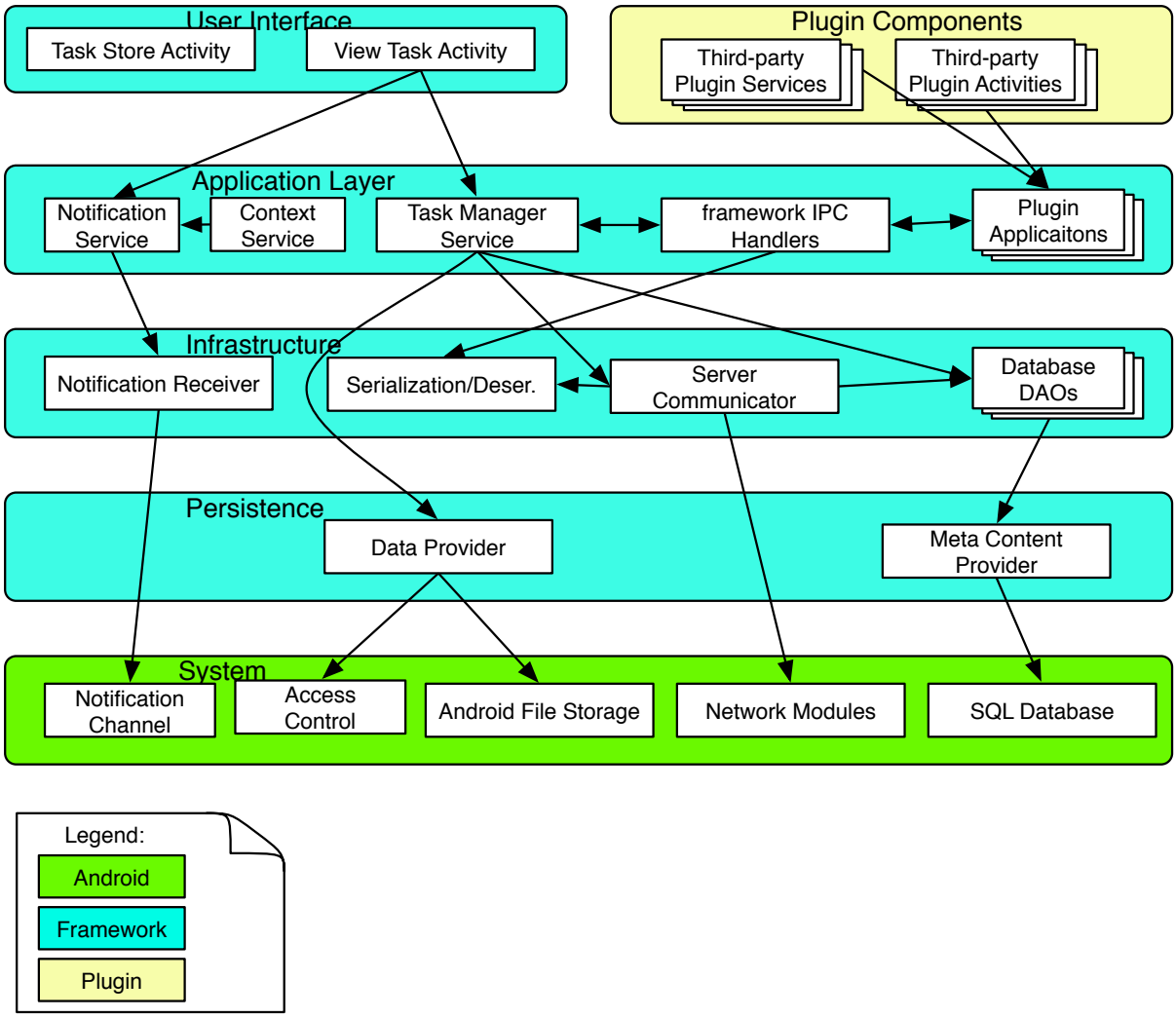

Figure 3.2: Client Architecture

The User Interface layer contains only visual representation meant for the user. It does not contain any functionality other than what's needed for interacting with lower layers and presenting information to the user. The user interacts with the UI to see, for example, what the available tasks are, view their descriptions and obtain information on the tasks' progress.

The framework's main funcitonality is in the layers beneath the UI layer. The `Task Manager` is the service that orchestrates behind the scenes the entire task execution and acts as a mediator between other components.

There are several modules meant for creating, accessing and modifying both meta information and generated data. The data model used by the modules has been discussed separately in §3.2. In terms of the modules' purpose, they provide different levels of storage abstraction in order to meet the various needs of the `Task Manager`, the UI, the `Context Service`, and the `Communicator`. The different `Data Access Objects(DAOs)` are abstracted the highest; they work on a `Task`, `Stage`, and `Stage Parameter` basis as specified in the data model in §3.2. They are to be used when there's a need to work with an entire entity such as a task or stage. The `DAOs` are most needed by the `Task Manager` and `Communicator`. The `Meta Content Provider`, on the other hand, is a relational database abstraction which allows for a more fine-grained access to the above meta information. Such access is useful to the UI which has only limited screen estate on which to display information. The `DAOs` use the `Content Provider` internally and wrap around the provider's relational interface to provide application-specific data access. The `Data Content Provider` handles file storage. It is used by the `Task Manager` to create and

access files in which any kind of stage-produced data is stored, and to also grant permissions to other stage applications to access it.

The `Server Communicator` manages the entire communication with the server. It has knowledge of the server's endpoints and their API. The `Communicator` is encapsulates the knowledge of how the data needs to be presented to the server and the form in which it is retrieved. The `Communicator` transforms this representation back and forth using parsing and serialization and deserialization techniques and then relies on the `DAOs` to update the client's data model. Because of this, no other module has any exposure to the server's implementation (and even its existence). All modules excluding the `Communicator` only work with the local data model. This encapsulation allows for communication improvement techniques to take place such as data caching with transparency to any of the other modules.

The `Notification Service` and `Notification Receiver` handle broadcast notifications that can originate either from the server or from a `Context Service`. These notifications can then be presented visually to the user. Lastly, the `Context Service` is a background service that follows the current context of the client and which of the registered tasks with their current stages have contextual requirements that are made possible.

Some of the client's modules need to be discussed in greater detail which is done in the following sections.

3.5.1 Managing Stages

One of the problems discussed in the reviewed literature that is present with ad-hoc crowdsourcing solutions is the interference such applications can have to one another and their compounded impact on the user experience. Following the design principle of creating a framework solves this problem because of the centralized control. In the designed system this is done through the `Task Manager`. It is the single point in the system that orchestrates the entire control flow of multiple tasks and gives permission for data flow to happen between the framework and a plugin. This organization makes it easy to fulfill the design goal of putting the user in control — allow the user to control the `Task Manager` and thus control the stages. The `Task Manager` is meant to run as a background service, however, so the user actions in the UI are to be forwarded to it, not directly applied. The module performs three main actions. They are all assigned to it because each one of them affects its internal state which the module needs to maintain and hence having them in altogether makes the design it more cohesive. The `Task Manager` facilitates *signing up* for tasks by delegating work to the `Server Communicator` and adjusting its own state accordingly once the server response is received. The module also handles *resets* of tasks and stages. The reset is made as a step-wise process, working on the current stage in a task, and then going one step back, and so on until the beginning of the task is reached. Resetting a stage involves not only resetting the plugin process but also updating the data model, cleaning up any produced data, and optionally notifying the server. The last and most important functionality of the `Task Manager` is to interact with plugin stages — *control* their lifecycle and obtain status updates. The details of the interaction are presented in the next section. However, in order to perform such action, the `Task Manager` needs to keep track of several things in its internal state.

The module needs to maintain an up-to-date information of what stages the user is registered for and which are currently running. The `Task Manager` must to do this for several reasons. Clearly, such information is needed so that the manager can actively communicate in the background with the running stages. Such information also puts additional security in place because any plugin that is started will immediately report (discussed in the next section, too) its existence to the `Task Manager`. If the manager has not allowed the execution of the plugin (which in turn means the user has not allowed it either), then the manager has the option to kill the plugin process (as the current system does) or to ignore it. Tracking the active stages allows the manager to maintain the overall `Task` progress. Since it is the `Task Manager` that follows the workflow specified by the task meta description, it needs to know what is the current step in the workflow, what is its progress and if there is a next step. The module does not provide an interface which allows the user, through the UI, or in any other way, to execute an arbitrary stage in the task's workflow. This

would be wrong as some stages will depend on the data produced by others and those may not have executed yet. If arbitrary execution is allowed, this will lead to the participants providing corrupt data, if any. Instead, what the `Task Manager` exposes is a functionality to begin or continue the execution of a `Task` from where it last stopped. As discussed in §3.3, the *control* flow is linear and hence the `Task`'s progress can be maintained by a simple pointer to the current executable stage. This pointer is to be incremented once the current stage is finished and is successful. The task is complete if there are no more stages to perform, and the last one was successful. In cases where there have been failures, the user can rollback the overall Task progress through resetting the stage's progress.

Finally, the `Task Manager` needs to have the most recent information of the active stages in order to manage their lifecycles. Such management is needed to ensure that when a plugin stage is no longer needed to be running, it is fully shutdown and that way the smartphone's performance is maintained as well as possible. While the underlying operating system also performs process management, the framework can be more aggressive since it has additional information about the plugin stages and knows their intended purpose. Furthermore, a need for lifecycle management can arise when the user starts a stage of a given type, performs some work, and then switches to another task and starts its current stage which happens to be of the same type. Such interaction is likely to happen often and hence is part of the framework's goals to allow it. The Android implementation presents some complications which are discussed in the Implementation chapter. Here, however, it must be pointed out that in such cases, the `Task Manager` needs to ensure both instances of the same plugin application are in a sound state which is a function of the task they are part of, the input parameters, and the progress achieved so far. The soundness can be achieved by either by relying on the plugin to manage the multiple stages, or by letting the framework do this and have the plugin only focus on its job with a particular instance of parameters. The second option is much more convenient for third-party developers and more appropriate given the wider view of the tasks the framework has, compared to a plugin. Hence, it is the second approach that is chosen.

3.5.2 Extensible Stage App Library

By design, the `Task Manager` does not distinguish between different types of executable stages in terms of how it interacts with them — the type is not of importance to the operation of the manager, given its responsibilities discussed in the previous section. Each plugin application is thought of as a separate from the framework, self-contained process. The process, however, should not be independent but controlled by the framework. To this end, the manager communicates with and orchestrates the stages through a pre-defined interprocess communication protocol (IPC) that is part of the framework and is provided as a ready implemented solution to the third-party plugins. This provision is done through a self-contained library that can be deployed in the plugin application during development. The library is minimal — it hides all the complexity of interacting with the framework application and only requires to implement a couple of straightforward functions, such as reporting the stage's progress as a fraction number. These are discussed in more detail at the end of this section. In exchange to committing to the interface (and thus the protocol), the plugin becomes available to the framework and hence useful to all its users.

As it can be seen from the client architecture diagram (Figure 3.2), all of the components of a plugin application — any UI and any background services, are considered to be in the highest layer of the framework's architectre and hence expected to change often and be of various form. These are the framework's hot spots. But it is not them that the framework directly interacts with. Instead, there is a single point through which this happens. All plugin components depend on this single point to obtain and relay information from and to the framework, and are also controlled through it. This point is the stage application.

The stage application is a singleton that holds the entire state that is *relevant* to the framework — any information about the plugin's progress, a collection of the received and parsed parameters, any dynamically received input, and most importantly — it has an implementation of the IPC protocol which the `Task Manager` also uses. The stage application is a singleton because from the framework's perspective each stage is thought of as a whole, a unit that has a single, well-

defined purpose and the singleton is the logical entity that embeds all the functionality needed to achieve this purpose. Concretely, this means that each and every one part of the plugin application is actively working towards the fulfillment of this one goal; all of these components are children of the stage application singleton in the plugin's logical hierarchy. They should only exist as long as the singleton exists, and should only work towards furthering its progress. Note that, as shown on the Figure 3.2, the application singleton is shown to be part of the framework. This is so because it is essential for the control and information exchange to happen according to the well-defined communication protocol and therefore the implementation should be provided by the framework. However, the information provision of the stage's progress, how stage progress is internally coordinated between the plugin's components (for example, exposed global state or an internal interface), and how some of the control orders are passed along to these components should be left to the plugin developer.

The provided IPC protocol facilitates control and information exchange. Its implementation comprises of several parts and the overall design is presented in Figure 3.3.

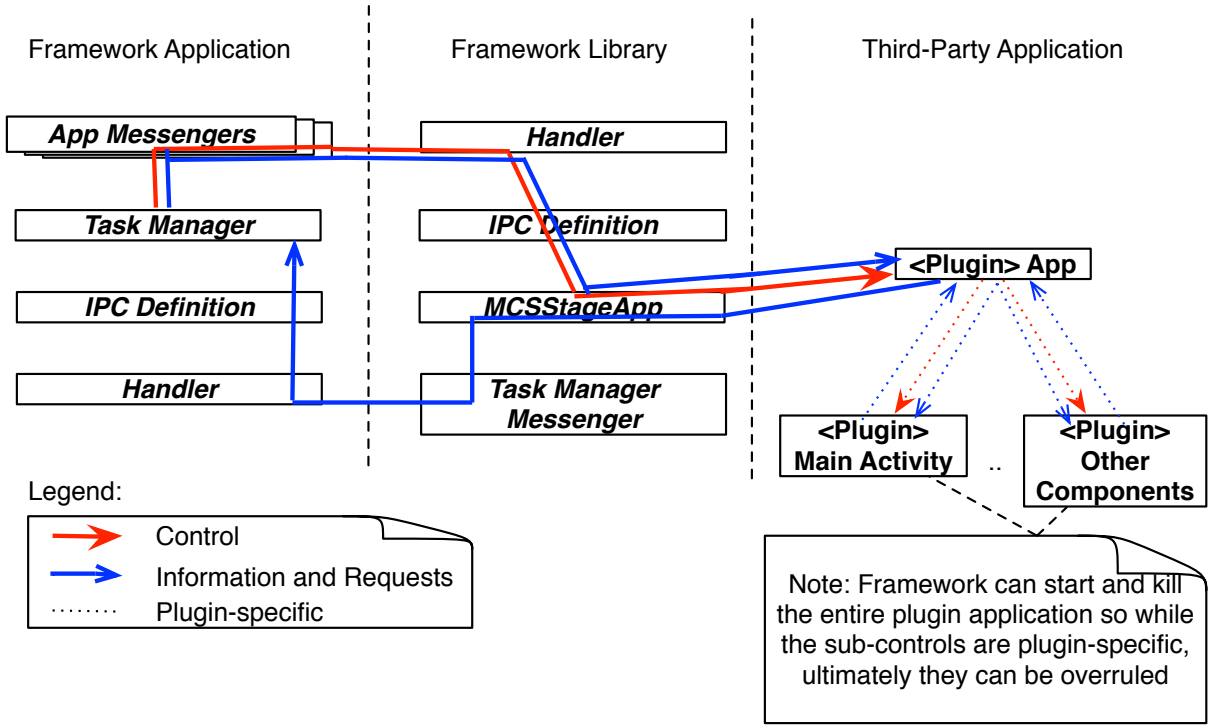

Figure 3.3: Role of the provided library in the control and information exchange between the framework and third-party stage applications

Each stage application has a message inbox which receives messages and handles them sequentially – a `Handler`. The messages contain a minimal representation of the communication type (constants defined in the protocol specification) and any additional data needed. The `Handler` then dispatches calls to appropriate functions in the module to which it belongs to — `Task Manager` or the stage application singleton. Similarly, to initiate communication, a messenger handle is maintained to the other side and the messages are sent through it. In the case of the `Task Manager`, such handle is needed for every active stage. This is achieved through the internal functionality (hidden from the third-party developer) of the provided plugin library. Upon initialization of the stage singleton, the process will bind to the `Task Manager` and offer its own communication handle. This approach has a few benefits. One is keeping track of the handles. Another is that such binding acts as an immediate notification to the `Task Manager` that a plugin stage has been started, and the manager can react accordingly — send a begin message once the initialization is done, or kill it, if it has not been authorized by the manager. On the plugin application side, the library provides an API to communicate progress or make requests (for storage

or data) to the manager while hiding away all the details of the communication. The IPC protocol and the functionality it offers has been summarized in Table 3.1.

Communication Type	Initiator	When	Payload Init	Payload reply (N/A if no reply)
Status	TM/Stage	Occasional timer/upon Progress	None/Status Data Structure	Status Data Structure/None
Create	TM	TM starts a Stage	Stage Parameters, Extra Input	N/A
Bind	Stage	Initialization	Stage Own Receiver	TM Own Receiver
Ready	Stage	Initialization done	None	N/A
Begin	TM	Stage is ready	None	N/A
File RQ	Stage	Need to store data in a new file	number of files and MIME type(s)	Uri(s)
Data Access RQ	Stage	Need access to data	task and stage ids that produced the data	Uri(s)
Kill	TM	Stage done or not allowed;new stage same type is to be started	None	Status
Finished	Stage	Stage complete and self terminating	Status	N/A

Table 3.1: IPC protocol between Task Manager (TM) and Stage applications

With the IPC protocol implementation in the library, the plugin application using it has access to the framework and through it — access to data that has been produced by other tasks and stages, given that the user grants access permission. In order to use the library, the plugin application needs to implement the required interface which is minimal. Most of it deals with obtaining information in a format that the framework can present to the user — for example, a textual and numeric representation of the stage's progress and status (*running, finished, successful*). There are only two requirements that are more demanding development-wise — the plugin must parse the parameters it has received, and it also needs to keep track of its produced data which knowledge is later to be transferred to the `Task Manager` (note, only the `URIs`, not the data itself, are passed, since the stage plugin never owns the data — it is only given permission to write it to the file storage). However, both of these requirements are something quite necessary to do anyway, regardless if an application with such specifications is to be written as an ad-hoc one or a plugin implementation to this framework. Lastly, parsing the input parameters is ultimately controlled by the plugin developer who, when registering the plugin with the server (discussed in the Server section – §3.6) declares what kind of parameters are required for the functioning of the stage, and what can be optionally supplied. This declaration is then imposed by the server when specifying new `Tasks`.

The precise interface implementation of the library is shown and explained in §4.2.1.

3.5.3 Notification Service

One of the design principles the project follows is trusting the smartphone client to self manage the task execution. However, there are use cases where there will be a need for dynamic input from the

server. For example, interleaving classification of machine learning algorithms with human input. The `Notification Service` module shown in the client architecture diagram (Figure 3.2) is a scalable solution to this need. The service is designed to be used for 'push' notifications. This is a common communication pattern that comprises of the client registering itself to the server and when something of interest happens on the server, the server notifies every client that has subscribed for such events. Push notifications are suitable for the design goals of this project and are preferred over its alternative, polling, for several reasons. One such is that the clients need to know about the event as soon as possible to enable some of the use cases found in the literature, such as providing further answers to *subsequent* queries of blind people [3]. However, such dynamic events do not happen very often and are unevenly distributed, which makes it quite unfeasible for a resource-limited device such as a smartphone to constantly poll the server for updates. Furthermore, polling is not scalable, because a large number of such update requests will keep the server frequently busy without any added benefit.

It is worth pointing out that the `Notification Service` is meant to work on a *framework* level, not task. All server notifications are to happen through the same channel. The `Notification Receiver` listens to the channel for notifications directed at the framework. It then passes such to the `Notification Service`. The `Notification Service` verifies the soundness of the notification, and handles it depending on its type. The notification is then presented to the user in a standard for the *framework* way, with the only differentiating part being the information shown in the presentation. This part is based on the Task and Stage to which the notification refers to and is obtained from their meta description. Finally, it is then up to the `Task Manager` to process the notification and to appropriately pass along the sent payload to the correct process. The way it is processed by the manager is identical to which a user pressing the start button in the UI — this is intentionally so because interacting with the notification is intended to have precisely the same user–smartphone meaning as pressing the button. Reusing the same functionality also makes for more streamlined `Task Manager` interface and internal implementation.

The kind of notifications the `Notification Service` handles is starting a stage with new input and canceling previous notifications before they have been interacted with (for when the server no longer needs such work to be done).

3.5.4 Context Service

In addition to dynamic input from the server, the `Notification Service` can be leveraged for an extra use. In its conceptual design, the module is meant to signify to the smartphone user there has been a change in some state which affects one or more of the tasks to which the user has signed up for. The will user in turn interact with the UI and consequently — the `Task Manager`. This purpose makes the `Notification Service` suitable not only for handling dynamic server input but also for reacting to changes of mobile context. As discussed in the literature review, mobile context is crucial to obtaining more accurate data from the clients [16], [8]. The contextual information is provided by the mobile operating system. It broadcasts context-related information upon contextual changes such as the enabling of the wireless antenna, a GPS location update, or a change in the user's activity. Any application is able to receive and react to these messages. However, just like the need to manage multiple tasks active at the same time, there is a need to manage the mobile context centrally. Otherwise phone performance and usability will be heavily influenced because every task for which the user has signed up for will need to have its current stage plugin application *running and listening* for such events at any one time, not only during the stage execution. Furthermore, it will require that the third-party developer implements this functionality which should not be needed given that the possible combinations of mobile context are not that many, and they are independent of the particular stage type. In other words, many stages are likely to have similar, often overlapping, contextual requirements (including none, too). Such overlap only makes the case for a single centralized context handling even stronger because there notifications about the same contextual item need to be handled only once no matter how many tasks are affected. Hence the designed `Context Service` in the framework. The `Context`

`Service` provides a simple interface of verifying whether a `Stage` is within the right context, which in turn can be used for deciding if it can be executed and also whether it can still continue to execute. This is achieved through using the `Stage`'s meta description that contains the contextual parameters (as shown in the data model in Figure 3.1). The other, and in fact main, purpose of the `Context Service` is to listen to all system broadcasts about context that are currently relevant. This relevancy is determined by combining all the contextual requirements of all the tasks that the user is currently registered for. This the module maintains as an internal representation which is actively updated upon task registration and progress.

Upon any system broadcast with relevant context, the `Context Service` verifies if any of the tasks' current stages have become applicable and interacts with the `Notificaiton Service`. The `Context Service` creates a notification for the user to see, letting them know that a stage's execution is possible, using precisely the same interface that the server would use to create a notification with the needed information.

From a end-user point of view, both dynamic input from the server and contextual updates, signify an opportunity to participate in a `Task`. Pressing the notification should let them do precisely that. Therefore, there is no need to for different kind of interaction once a notification has been created. The internal handling from the UI and the `Task Manager` perspective is entirely the same, regardless whether the user has pressed a *Begin* button, the server has sent a notification containing new input, or the `Context Service` has created a notification to start a Task's current stage.

3.5.5 UI

The User Interface is intended to address a few precise goals. The UI needs to show the user the most up-to-date information regarding the available tasks on the server, and also the current progress of any tasks to which the user has signed up for. Such freshness of data is essential because the framework will be managing several ongoing plugin stages in which the user is participating — for example, recording sensors for one in the background while using the camera to capture video for another. When displaying each of the task's description to the user, any progress that they have made and that has not been reflected on the UI will only confuse them and could lead to erroneous data being produced or, more likely, their involvement to stop.

Such UI responsiveness is achieved by requiring that the data model, discussed in §3.2, is updated immediately upon any progress. This means that the `Task Manager` uses the information received from the third-party plugins to combine it with its own knowledge of the overall task progress to update the status of the tasks. This is done through delegation to the data access objects services. Then, it is for the different parts of the UI that are currently visible to subscribe to parts of the data model (for the specific tasks and stages displayed) and listen for any notifications about data changes upon which the data is reloaded.

3.6 Server

The framework's mobile counterpart is a intended to be a cloud service — an always on, networked server which provides its functionality over the Internet. The purpose of the server is to collect the data produced by the mobile clients and organize it. The server also manages user registration and participation in tasks which the server makes available to the users. Lastly, the server is meant to allow the creation and distribution of tasks, and the execution of the parts of the tasks that are to be run on the server, such as training a machine learning algorithm with user-obtained data.

3.6.1 RESTful web services

Following the design principle of building a framework, the server's design is to be made extensible and comprised of components with a single, well-defined purpose that can be flexibly used. Web services are very suitable for such a goal as they are meant to be completely self-contained and they provide external interfaces to be interacted with. Any user-facing functionality is to be achieved

with a thin client that wraps around the necessary web services. Such design decouples the server's responsibilities into modules that can be individually modified without impact on the rest of the system. Web services offer better opportunity for scaling and also make it easier to be orchestrated.

When designing the server architecture, a decision had to be made about the choice of kind of web services to be used – RESTful or WS-* (SOAP) (briefly outlined in §2.4.2). The choice between the two is a design, not an implementation one to be made; the decision sets a design style in place which affects the overall server architecture and how its components are used. Simply put, WS-* architecture would have less components but each of them will have more verbs (i.e., richer and more complex API). The focus will be on the interface semantics and the complex stateful interaction with the components. On the other hand, a functionally equivalent RESTful architecture will focus on the data representation. It will have more services representing the different kinds of Resources that need to be exposed and the available API on them will be limited to CRUD(Create, Read, Update, Delete) operations because of the HTTP protocol (most often used with REST services) which only allows four operations — GET, PUT, DELETE, and POST.

RESTful web services were deemed a better choice for the project's purposes compared to WS-* services for their simplicity, performance, and lightweight communication. A REST approach guides a design where communication with the web service is stateless. Designing for stateless communication forces building services that interact in one step. This is useful when dealing with mobile devices because their limited and unreliable connectivity may cause inconsistency problems. Furthermore, the framework's functionality focuses on data handling much more than on computation, and it only involves simple CRUD operations on data – for example, collect data via the client application, perhaps process it, and then upload it to the server. On the server, the data can be used to update the state of a machine learning algorithm. There are no critical parts that require, for example, a 2-phase commit transaction of the data, or a complex interaction between the client and the server. In such cases, SOAP web services would have been the better option. For this framework, however, the interaction is a simple data transfer. If an operation fails, then it can be safely redone (given its not a POST operation where additional internal functionality will be needed) because of the idempotency of the GET, PUT and DELETE methods. The safety and idempotency of these operators needs to be ensured by the designed web services. Lastly, stateless communication improves the web service performance on the server side since, by definition, there is no state to maintain. The sole purpose of the server becomes the to generate appropriate response for each received request.

RESTful web services are quite suitable for mobile devices for two more reasons. One is that the web service has the freedom to represent the data in a format that the developer chooses, which means that bandwidth and resource requirements can be reduced with an appropriate choice. This is in contrast to SOAP web services that enforce using the verbose and slow to parse (compared to other solutions) XML. The other reason is that REST responses can be cached and hence the communication between the client and server completely skipped. This is beneficial not only for the mobile device but also for the server's load. Overall, RESTful web services a more lightweight and simpler to interact with, which benefits the mobile client.

3.6.2 Resources

The overall server architecture is presented in Figure 3.4. The server exposes several resources as a RESTful API. With the exception of the `Machine Learning` and the `Client Notification` resources, the resources deal with simple CRUD operations which follow the data model presented on Figure 3.1. Each of them is backed up by a `DAO` which in turn either abstracts away a NoSQL database or a filesystem storage.

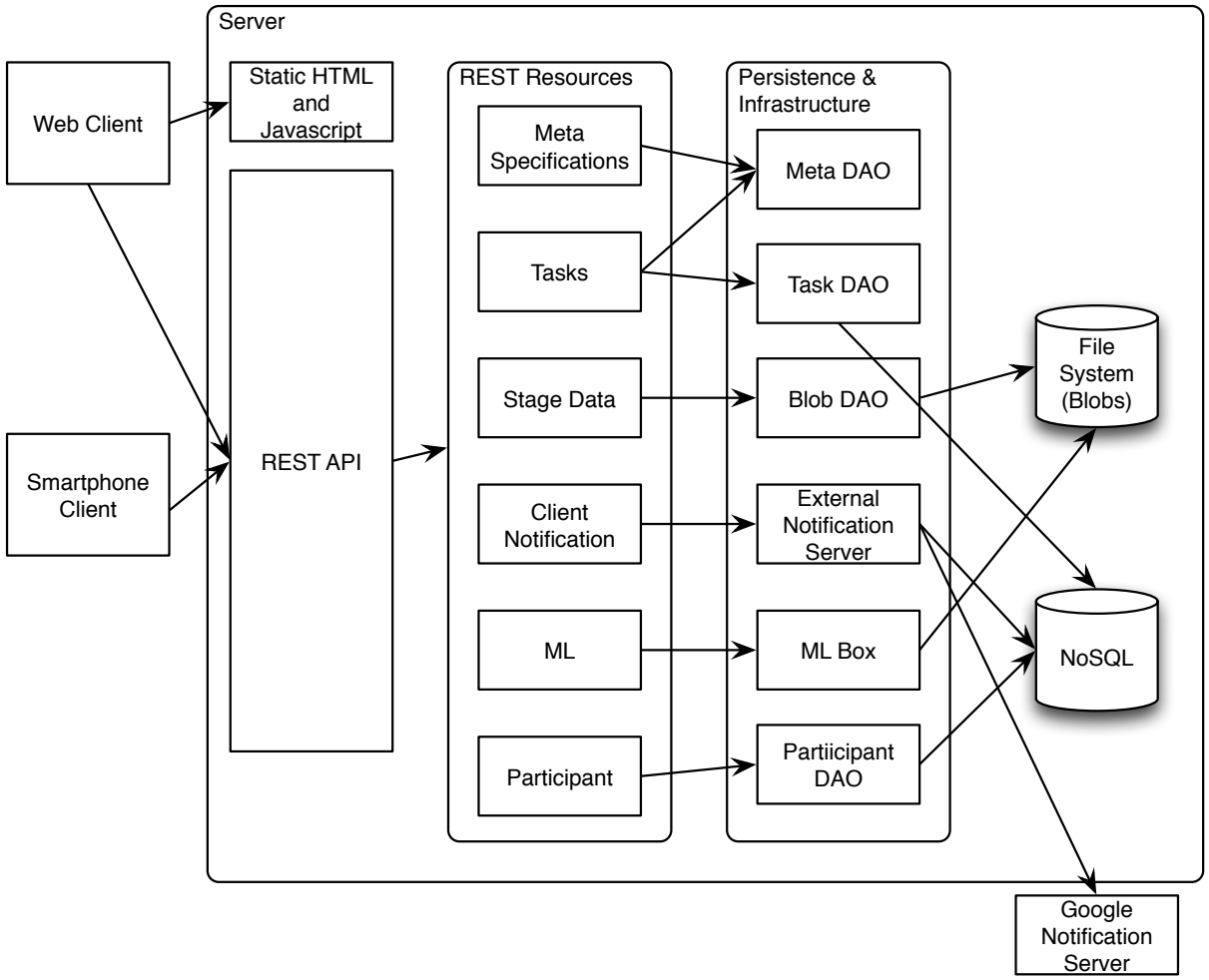

Figure 3.4: Server Architecture

The `Meta Specifications` is the resource representing the description of plugin stages. This includes the required and optionally accepted parameters needed for the execution of the described plugin. The resource also contains the unique name of the `Android Intent` action that the framework application should use to invoke the plugin. The resource is to be used by third-party developers when registering their new plugin. It is also to be used by the task creation client through which new tasks are to be submitted. The client's UI can be dynamically populated depending on the stages the user selects to specify in the new task. The client can then locally verify that the provided input parameters meet the plugins' requirements. If the test passes, then the representation of a `Task` resource is sent via POST to the `Tasks` resource.

The `Tasks` resource is a collection of children `Task` resources. It represents all the `Generalized Task` meta specifications that have been input in the system. Upon receiving the POST request from the task creation client, the `Tasks` resource also verifies the input after which it creates a full representation of a `Generalized Task` that reflects the data model shown in §3.2 and stores it using the `Task DAO`. One important part to note about `Tasks` resource and the `Generalized Task` class that it represents is that when the mobile client asks for the available tasks, all the `Generalized Stages` that the task contains that are not meant to be executed on the mobile client will not be shown to the user. Instead, the stage will be replaced by a generic one which can optionally block the user from progressing further in the task until notified by the server through a notification in the background. This is done to allow for workflow in which the task requires that the server finishes performing some work before the user carries on with their part of the task.

`Participants` represents all users and contains information of their involvement in tasks and the ids of their devices which can be used to send notifications to.

The `Stages Data` resource represents the data produced by a given stage. It organizes this data per user which is useful for crowdsourced machine learning and also potentially for controlling access to the data on a per user basis.

The `Client Notification` resource represents a web service able to send notifications to all participants of a task who are at a *specific* stage in the task execution. The web service ensures that a notification is sent to all of the tracked devices of a participant. The service leverages an external service — Google's Cloud Messaging. It does this by sending a POST request with the data to be sent to participants, along with their device ids.

The design of the `Machine Learning` resource requires more attention because of what it represents. Externally, the ML resource is another RESTful web service. What is represents, however, is not a simple structure of data but a machine learning algorithm. The algorithm's data model can be seen on Figure 3.5. An algorithm training is initiated by executing a POST request which contains the type (`ML Box`) of algorithm to be created, together with the URI of the `Stage Data` resource which is to be used for training and cross validation. The details of how the data is used by the `ML Box` to train the algorithm are of no concern to the framework, as long as the ML Box provides an interface to be *trained*, given an URI, to *classify* a new data point, and to report its *status*. These three actions are later used by the `ML` resource when generating responses. The POST request returns the URL at which the newly-created ML resource can be accessed at. In the meantime, the `ML Box` has begun training the algorithm in the background. In other words, the POST request is asynchronous. This is on purpose, given the long training time requirements that can be expected of a machine learning algorithm.

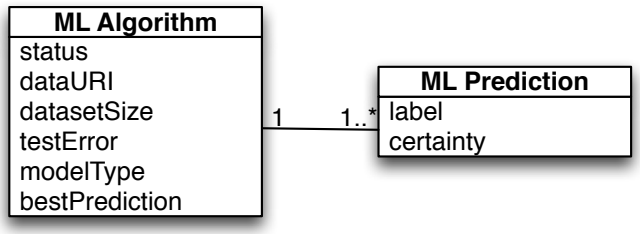

Figure 3.5: Data model of the ML Resource.

Performing a GET request on the new URL will return the representation shown in the presented data model. Note the attribute `status`. This signifies whether the algorithm has finished training, or is still in the process. This can be used by the `ML` clients to check when they can begin using the resource. Once done with the training, then it can be used for new data. This is achieved through POST requests to the new URL which contain the new data point. The received representation as a response is again the `ML Algorithm` object but both its `bestPrediction` and the collection of `ML Predicition` are non-empty. `ML Prediction` is a simple two-attribute class representing a label and the certainty with which the algorithm has classified the item to belong to the label. That way, whoever is using the web service can decide whether the certainty is satisfactory. To this end, the `ML Algorithm's` attribute, `testError`, can also be used. It holds the error measure obtained during the cross validation of the algorithm.

The presented server architecture is entirely comprised of web services that can function independently. For the execution of more sophisticated tasks that include operations not only on the mobile client but also on the server, such as interleaving machine learning and mobile crowdsourcing, a module that interprets the task and orchestrates the entire execution is needed. However, because the focus of the project has been on the client, this module has not been created. Nevertheless, the simplistic nature of the RESTful web services and their modularity make it possible to have an execution engine which orchestrates such execution. In other words, it is not required that this module is part of the framework, given the web services' accessibility and generality.

3.7 Data Flow

With the client and server designs explained in the previous sections, here the overall framework usage is briefly described. This is done with the help of Figure 3.6 which gives a lenient representation of the expected data flow. The direction of the arrow signifies the direction of the data transfer.

The framework is designed so that third-party developers build their plugin stages, and register them with the server. A tasker interested in collecting data or training a machine learning algorithm, or combining both, will then specify a general task. This task, along with others, is what potential participants can browse through their mobile clients via interaction with the `Tasks` resource. When the users sign up for a task, they can begin executing the mobile part of it on their clients, and thus generating data during the various stages the task is comprised of. Some of the stages will be to upload parts of the generated data to the server. At this point, the machine learning resource can be activated by being given the URL to the stage data. The last step (seven) – training the algorithm, sending notifications to the user, and generating more data, is repeated until the task is completed.

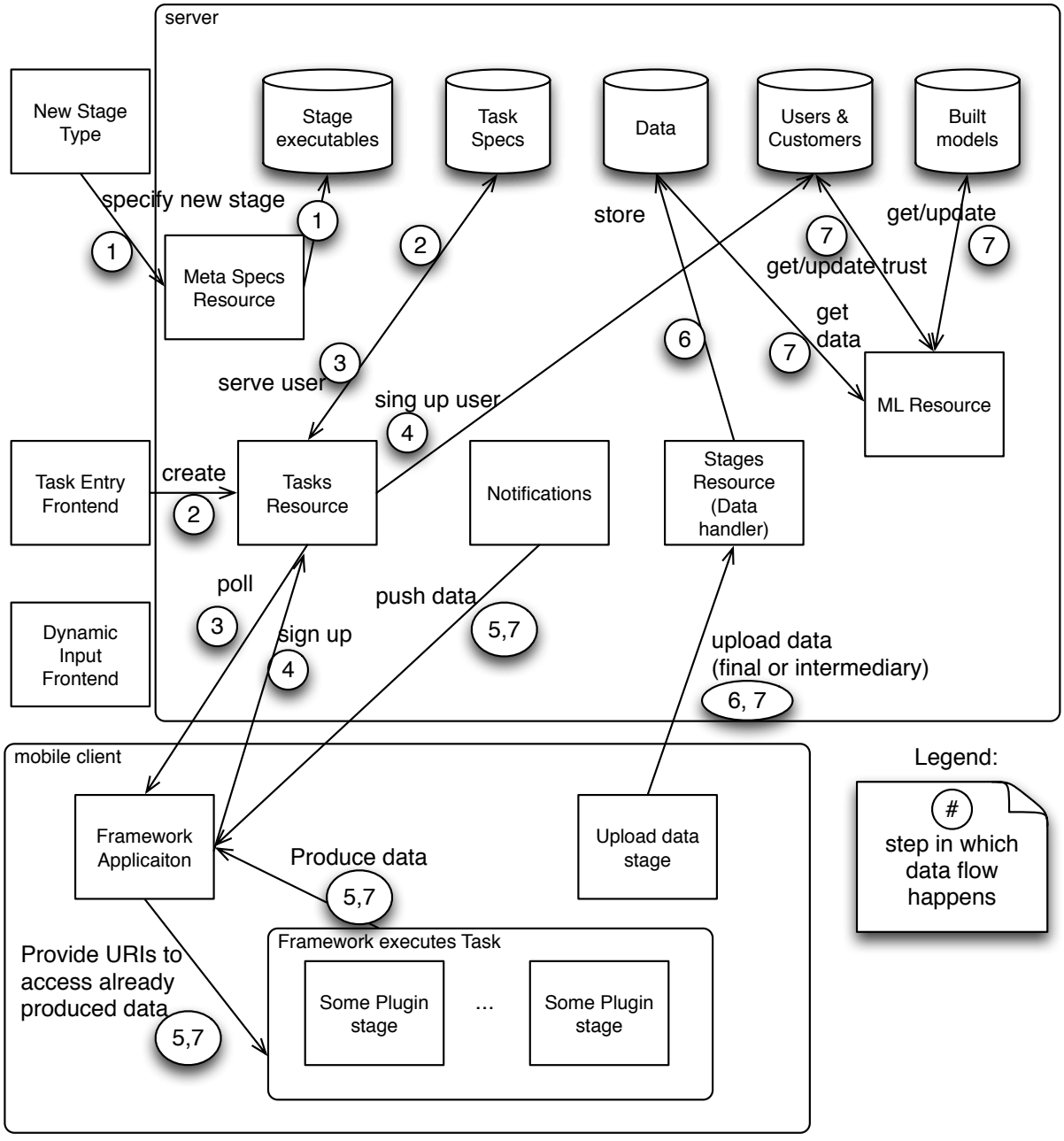

Figure 3.6: Conceptualized data flow.

Chapter 4

Implementation

This chapter discusses the implementation choices that were made, some of the problems encountered in the process and how they were solved while building the system as designed in the previous chapter. The system has been named *Hive*. When considering a choice between alternative implementation options (for example, libraries, development techniques, internal data structure representations), the choice allowing to follow the project's design principles the closest was deemed most appropriate. For example, performance in the form of responsiveness and usability are essential for the client; this consideration has strongly influenced the choices of a serialization library and the UI and communication implementation. The difficulties that were encountered and choices that needed to be made showcase the need for the existence of the framework this project has set out to deliver because they are not straightforward to address and can be quite time-consuming.

The project has focused on solving the challenges present on the client side of the targeted software. The client is meant to be a smartphone application acting as a framework to other applications — stages, which are to be provided by the framework or by third-party developers, and no distinction is to be made between the two. The Android operating system was chosen for the client's implementation. Section §2.4.1 gives more details on the relevant terminology and behaviour. It was selected for its rich APIs and broad popularity which can make the framework's adoption easier. Most of all, however, it was chosen for its known flexibility and freedom, *relative* to other mobile platforms, in allowing non-typical applications to be built — a framework app that treats other third-party applications as controllable plugins is certainly such a case. Establishing such control has been one of the key implementation challenges, even in Android. This is because the Android OS runs every application in a sandbox that completely isolates the application process from other processes' influence, and treats each application as an independent one. The programming language implicitly selected with the choice of Android is Java.

The server is implemented to run REST web services backed up with a file and a NoSQL database. Its implementation is in Java, its web services are offered via Java Servlet containsers using Jetty and running on Google's App Engine platform. Additionally, some amount of HTML code and Javascript was written to build the task creation web client.

4.1 Main Application

The implemented application closely follows the architecture presented(§3.5) in the Design chapter. Its classes have been grouped in packages that approximate the modules in the designed architecture. The only missing part in the system is the `Context Service` which, due to time limitations, was not implemented — there are `DAO`, data structures and interfaces put in place for its support but there was insufficient time to tie everything together for a working service.

Interaction between the system's modules happens through interfaces. Unlike normal Java development, however, this is not done through having an object reference but instead through messaging (`Intents`) between `Components` of the application. It is crucial that this difference is well-understood and respected during development because it is within the core nature of how Android functions and affects the smartphone's performance and usability. Because it is the Android

system that handles the lifecycle of an application and its components, not the running application, it is not suitable to maintain such object references between components — otherwise, the system will either break the references when it has cleaned the components up (and the application will crash), or, if forced by the implementation, the system may never clean them up, even when they are not used and the system needs to free up resources. This second option affects overall phone performance and will have negative impact on the battery and usability. From the Android system perspective, it can be viewed as a memory leak. This can be a common pitfall for novice Android developers.

The minimum Android SDK version used is 15 (Android 4.0.3) which limits some of the potential users with older phones but ensures the framework built can make use of more robust and optimized APIs and techniques (such as HTTP response caching and `Loaders`).

4.1.1 User Interaction

The framework application provides the user with interface to view all available tasks on the server, see their description, and participate. The UI comprises of two main screens that are built using Android's `Activity` and a combination of visual elements available from Android. Android's design guidelines were followed and the static UI was created in XML while anything dynamic is done in code. The two main screens the user interacts with are presented on Figure 4.1. Note that each of the lists, one showing the tasks, the other – the stages a task involves, are dynamically loaded. The lists are directly linked to the underlying data model which, upon update of the precise URI of the displayed data, will notify any listeners of the change. The UI will in turn reload the data. This is done with the more complex to implement Android `Loaders` which are loading data from a `Content Provider` and subscribe to data changes. This is all done asynchronously and in the background. It is worth pointing out that the `Loaders` access only the `Task` and `Stage` meta information that is to be displayed, not the entire objects, ensuring there are no wasteful data loads and transfers. The choice of using `Loaders` in combination with the decision to implement `Content Providers` which are to be used in the framework is essential for the design goal of responsive and up-to-date UI.

Notice the refresh button in the `Task Store Activity` which works with the `Communicator` module, and the three buttons in the `View Task` screen — these correspond to the three actions that the `Task Manager` performs and are later detailed. In likeness to the `Task Store Activity`, another `Activity` can be made, reusing its functionality but showing only the tasks the user has registered for and are still unfinished, or tasks contextually possible at the moment. However, its functionality would be identical to the `Task Store Activity` — only the filtered data would be different. Hence, it was not given high priority to implement.

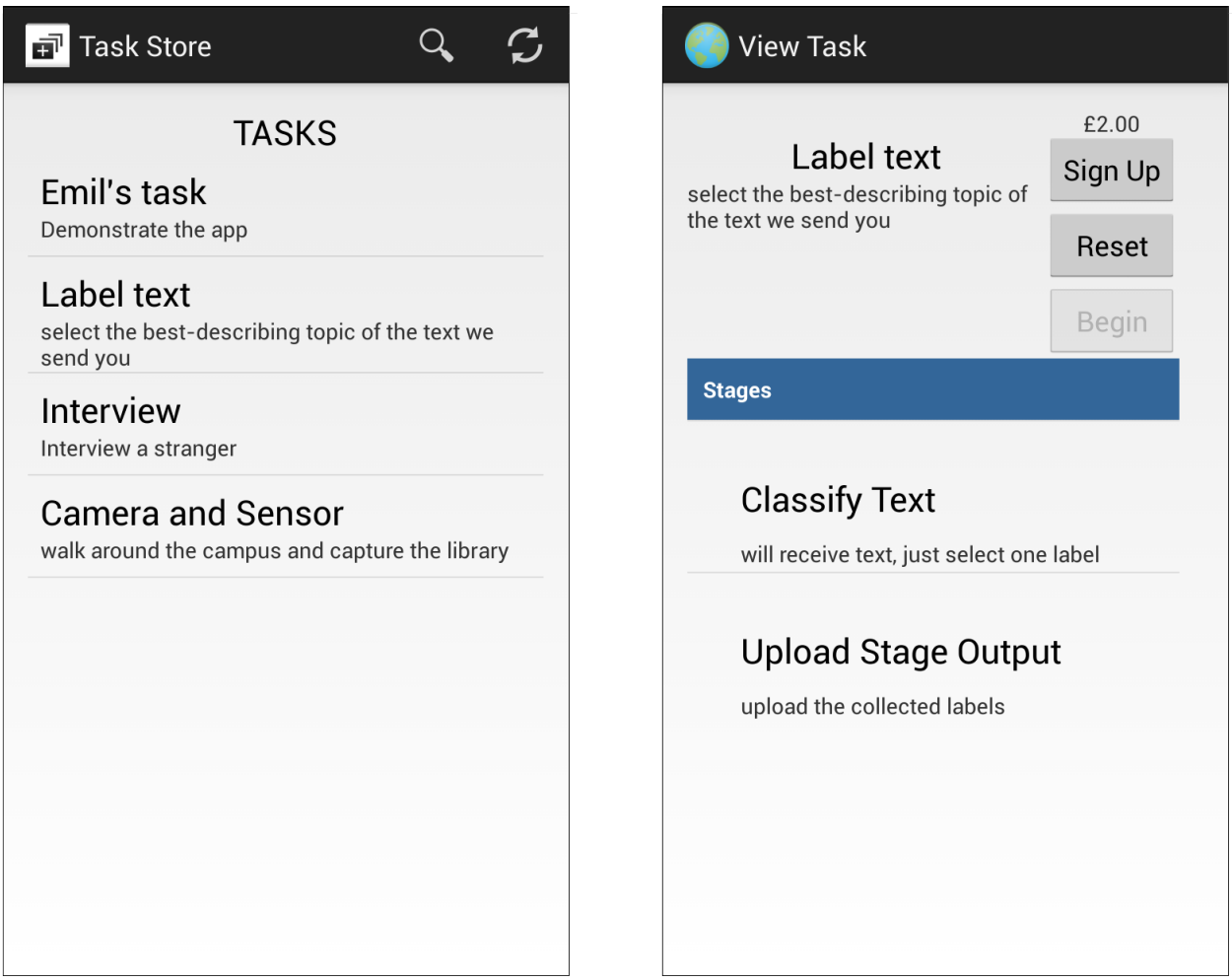

Figure 4.1: Task Store and View Task screens.

Before the user can sign up for and participate in any tasks, they need to register on the server with an unique user id, needed for several purposes such as machine learning and data privacy. The process also requires that the device itself is registered with the server (needed for notifications). Both actions happen in the most convenient way to the user — the user id is the user's Google account which every Android device has (if there are more than one accounts, the user is prompted to confirm which), and the device registration happens completely transparently to the user. This information is then sent for the server whenever it is needed.

The `View Task Activity` guides the user with the task's progress. As the user progresses, each of the stages receives a textual and graphical representation of their status. This is shown on Figure 4.2. The progress information is obtained from the stage plugin applications involved in the task — it is part of the interface that each plugin needs to agree to. For example, Figure 4.2 shows the progress of collecting sensor input for ten seconds from three sensors, capturing two images, and then uploading the five files produced by these stages. Notice how the `Upload Stage` had first failed. Such status is up to the stage plugin to decide. The user can simply retry the stage.

While the UI is not visually advanced, in order for it to work fluently and to be always fresh, a lot of work happens behind the scenes. This work is presented in the following sections.

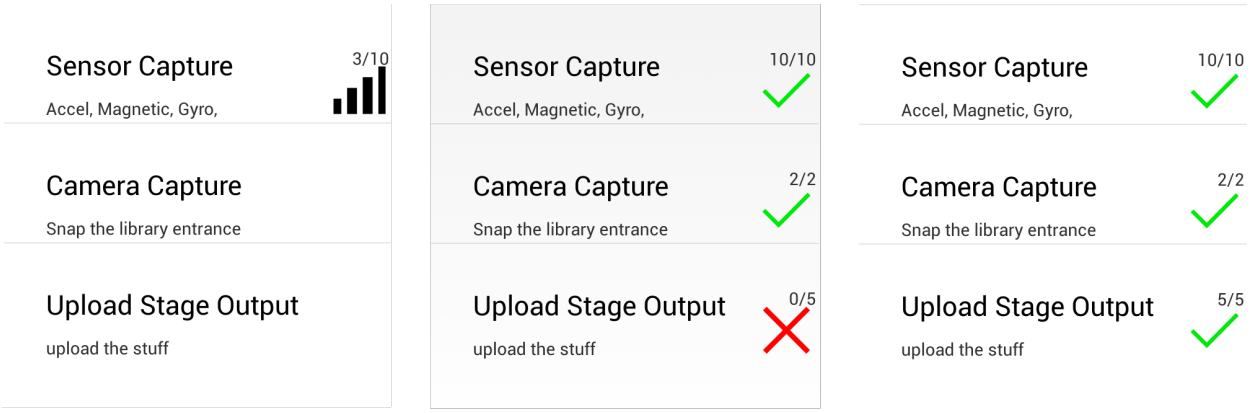

Figure 4.2: Visual and Textual progress feedback.

4.1.2 Storage & Data Handling

Data is retrieved from the server and also communicated between the framework application and the plugin stages. The data is complex and structured and hence it would be tedious and error-prone development to transfer it flatly. Instead, it needs to be serialized and deserialized as a structure. There are multiple ways to achieve this. One of the several reason for choosing RESTful webservices over WS-* has been the freedom to chose data representation format. JSON is a much more readable and compact format than XML which is the enforced format by WS-* services. Furhermore, XML is computationally more expensive to parse, regardless if relying on the DOM (Document Object Model), or SAX (Simple APIs for XML). Therefore, both to save bandwidth and computation time, JSON is the preferred option. This is essential for mobile phones where communication bandwidths are small and larger payloads result in the phone's radio being in *ON* state for longer which can have battery implications [13]. There is a plethora of high-performance serialization/deserialization libraries for parsing JSON. `Jackson` [6] has been benchmarked [10] to be one of the fastest (native Java and Android parsers included) library in many aspects. Jackson also has a binary transfer module which can also be leveraged if further optimization is needed. Considerations about serialization and transfer size were made to make certain there's minimum impact on the phone, given the expected frequent meta information transfers between client and server and between the framework and the plugins.

The storage modules involve a few classes that operate on different abstraction levels, meeting different needs. As per design, the meta information is to be stored in a relational database. Android provides a working SQLite implementation which can be set up through SQL queries. However, the database is not thread-safe and it is up to the developer to ensure this. This can become a problem when multiple instances of a `Content Resolver` — one of Android's ways of accessing data, are instantiated from different places and the data is updates. This has been solved by ensuring that the database instance on which the resolvers depend on, is the same. That way, if there are any collisions, only one will succeed and the data will remain in a sound state.

To ensure there are not any discrepancies between the database definition through SQL, the data model, and the `Content Providers` (which externally are seen as a relational database, too), several contract classes were created with all the constants that describe the columns in each table and the URI paths. The URIs are Android's way to navigate all `Content Providers` that are available on the device. Tables for `Task`, `Stage`, `Stage Context` were created for their normalized forms.

Once created, the SQLite database instance is obtained by the `Meta Content Provider`. When a `Content Resolver` from any application tries to access some data through providing a URI, the Android system finds the `Content Provider` to which the data belongs through this URI. It then verifies whether the accessing application has permission to access the content. The `Meta Content Provider` does not give permission to any application to access it, including the

plugin applications; it is visible only to the framework application. The way meta information is transferred is through copying only the relevant Stage information in the `Intent` that is sent to the plugin application.

All data changes happen through the `Meta Content Provider`. Furthermore, the provider operates with URIs which allow subscribing listeners to them. Therefore, it was most suitable to implement in the `Meta Content Provider` the functionality to notify entities which have subscribed to the given URI. Upon an *insert* or *update*, the `Meta Content Provider` will check for any changes to the database and if so, will send notifications. The URI has been made flexible enough (in the `Meta Content Provider` handling) to address a single entity or a group of entities in a given table, or across tables. It can be general enough to point to the entire table, too.

The `DBDAO` provides higher level abstraction. Rather than working with columns as the `Meta Content Provider` does, it operates with `Tasks, Stages, and Stage Contexts`. Internally, it still uses the provider to perform updates but the `DBDAO`'s purpose is to offer convenience methods for the `Communicator` and `Task Manager`.

In addition to holding the meta information, the framework application also *owns* all the data produced by any of the plugin applications. This is done both for putting the user in control of their data privacy and also for flexible data sharing between plugins that is not restricted by the plugins' lifetime nor their sequence order in the task. It is normally the case that data stored by an application in its own Android-provided storage is accessible only to itself. If there is data sharing to be done, then a `Content Provider` needs to be exported and other applications will have access to the data it points to. However, this results in a very coarse data access – all or nothing, which does not meet the framework's goals. Instead of exporting such provider, Android's `FileProvider` has been used. It has been extended into a `Data Content Provider` with additional convenience functionality. This provider grants read and/or write permissions on a *per URI* basis which is ideal for the framework — as long as the URIs are constructed in such a way that a URI signifies the output of a particular stage plugin for a particular task, then the data access can be tightly controlled. The URIs are constructed only by the `Data Content Provider` and have the form of $file://[framework\ constant]/[taskid]/[stageid]$. The permissions granted can be revoked at any time meaning that even the plugin application that created the data in the past will not have access permissions unless it requests it and the user allows it. The interaction is shown on Figure 4.3.

Additionally, the `Data Content Provider` handles requests for creating new empty files which are to be used by a plugin stage, or for cleaning up a stage's data output, if it is being reset.

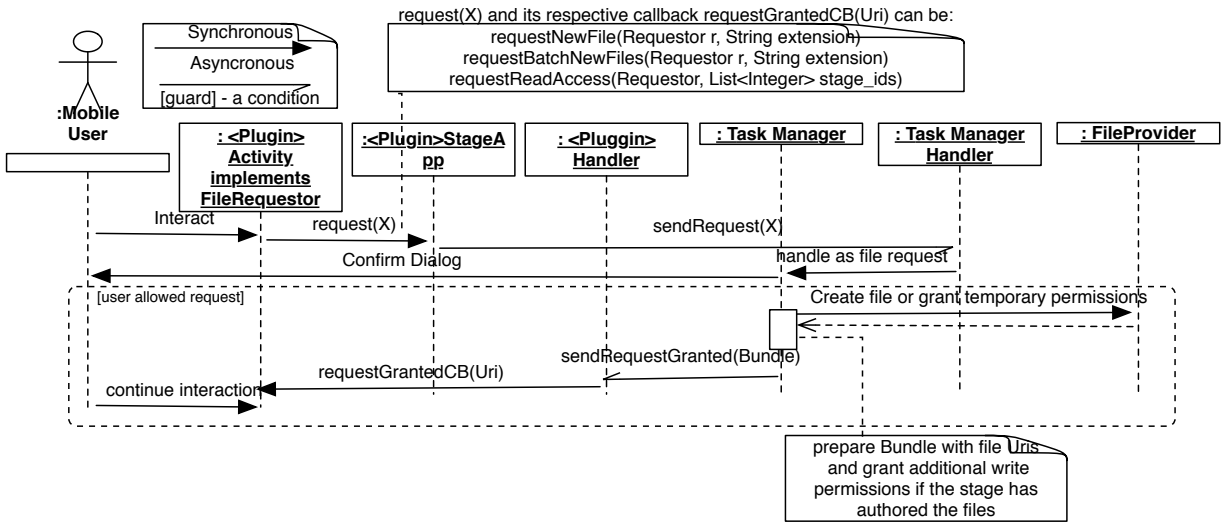

Figure 4.3: Sequence of a plugin performing storage access request.

4.1.3 Task Manager & Interprocess Control

As per design, the `Task Manager (TM)` is at the center of the framework's functionality. It is implemented as an Android `Service` that runs in the background for as long as the framework application is running. This is so because of its role as a mediator between all other modules.

The `Task Manager` internally keeps track of all the active (i.e., currently running, whether on the foreground or background) tasks. While it is tasks that are being managed, it is worth reminding that at any one point, due to the sequential control workflow design decision (§3.3), there is only one stage running per task. It is stage plugins – separate applications – that the `TM` controls. The `TM` does the tracking by keeping a `Map` with `ExecutableTask`, keyed by the unique task id. An `ExecutableTask` is an extension of the `Task` data model presented in §3.2 that has the additional functionality of 'knowing' when the task is completed, being able to track its current progress and hence, current stage, and generating the `Intent` that is needed to start the current stage. The latter two functions is what makes the class an Android-dependent one, unlike the plain Java object which is `Task`. The created `Intent` contains all the meta information that is needed by a stage application — i.e., the information stored in the `Stage` model. Any additional dynamic stage input from the server is also added. The information is serialized using `Jackson` and stored in the `Intent` which is later to be used to invoke a new process — the plugin application. There, the contents will be deserialized by the plugin library and made available to the plugin's components.

Additionally, the `Task Manager` needs to keep a `Map` of the *type* of active stages that are currently running. A stage's type refers to the application that is used to perform it. Android's model of invoking *specific* `Activities` (and as a consequence of that, its application instance) is through using `Intents` containing well-defined unique strings. It is this unique string that is used as a type indicator. The string is the action to which the primary component of the plugin application will respond to. This is defined during the plugin registration on the server side. The type information needs to be maintained for the case when another stage of the same type is started by the user and the `Task Manager` needs to handle the transition. How and why this is done is explained later in this section.

Lastly, the `Task Manager` keeps track of all the handles used to for interprocess communication with the active plugin stages. There is an outgoing handle for each plugin and its implementation is provided by Android — a `Messenger`. As described in §3.5.2, they are used to send messages following the framework's predefined protocol. The `Task Manager` makes it possible for other components to bind to it, and upon this binding, the components offer their `Messenger` handles. This is all done by the plugin library upon plugin initialization.

The way the `Task Manager` is interacted with from *within* the framework is through `Intents` that contain sufficient information to perform one of its three main duties regarding tasks — *start*, *register*(sign up), *reset* (curent stage). `Intents` make for a standard interface that can be used not only by the UI but also by the `Notification Service` and hence, by the `Context Service`. Each of the three actions involve a complex interaction with other application components, the plugin applications as well as internal work for the manager. The interactions are detailed in the following sequence diagrams.

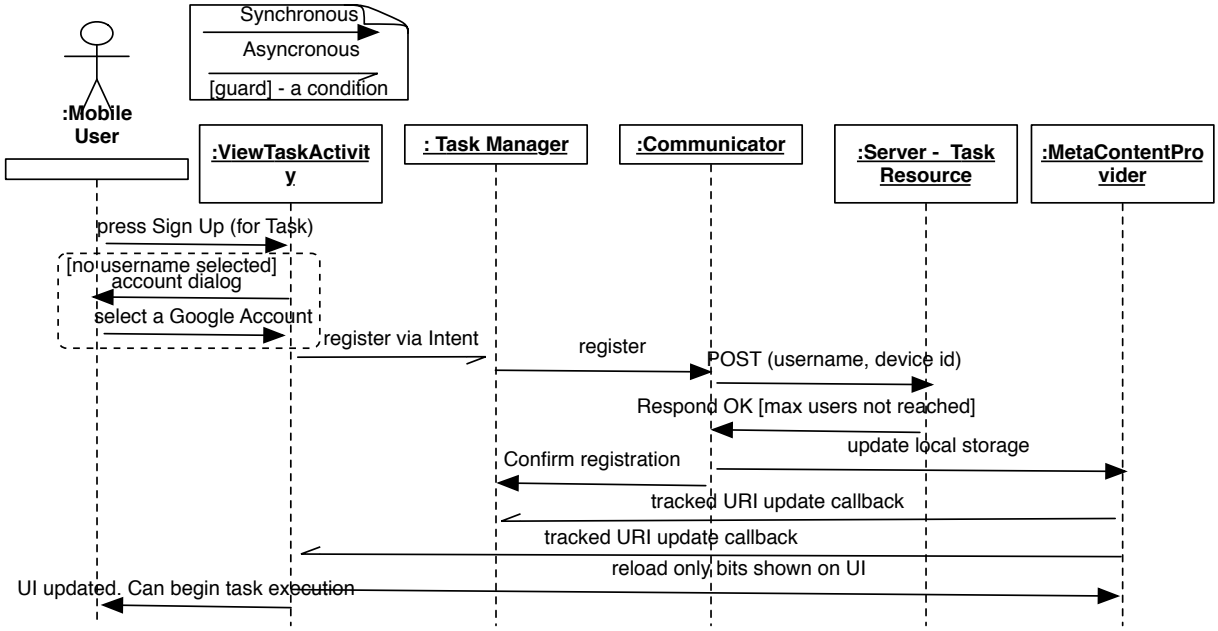

Figure 4.4: A user signs up for a task.

Signing up for a task involves interacting with the server. This is required because some taskers might want to restrict their costs (whether monetary or server load) and limit the number of users who sign up for a task. The sign up sequence involves ensuring the user has picked up a registration name, and using the `Communicator` to register the user at the server providing the user details. Centralizing the sign up for a task at the server ensures that the maximum number of users is always respected. If this was done locally and only later verified with the server, then there could have been discrepancies because of the delayed update. If the server allows the task sign up, then the `Communicator` will use the `Meta Content Provider` to update the registration column of the given task. This update will in turn trigger a notification to the UI because it is the same URI the `Communicator` uses that was also used to look up the task data from the UI. Hence the data will be reloaded. The UI can then adjust to the new values by making the 'Begin' button available.

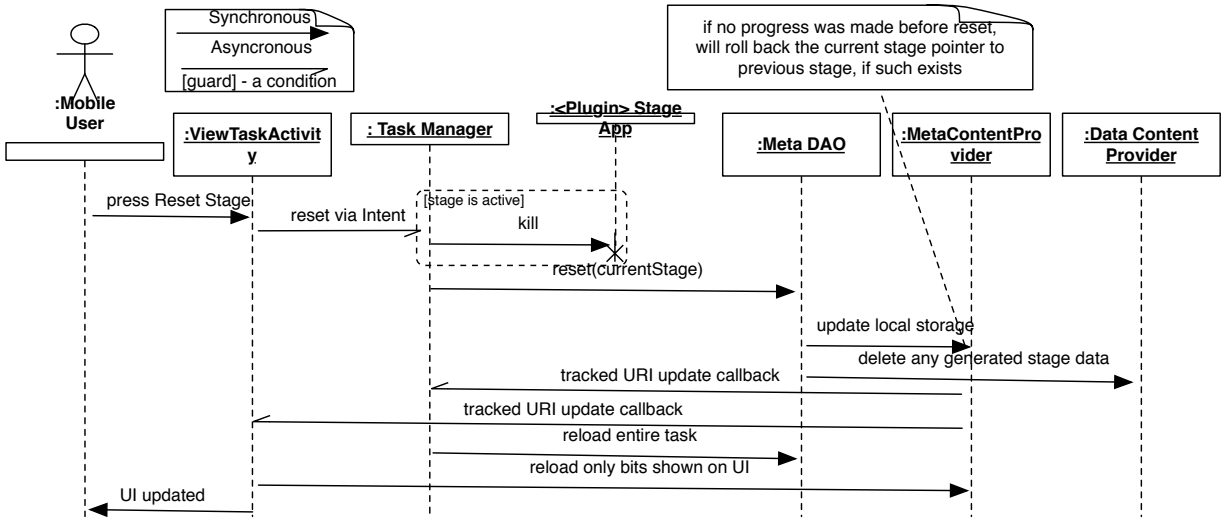

Figure 4.5: The current progress is iteratively reset.

The *reset* action is an iterative process which works on two levels. The first is on `Stage` progress — if the current stage of the task has achieved some progress, then this progress is reset.

Any data that was generated by the stage plugin (for this particular task only!) is cleaned up, and the stage's status is reset. If it was the final stage and was complete (hence, the task was complete) then the task's status is also updated. If, on the other hand, there has been no progress at the current stage, using *reset* works on a higher level — the overall `Executable Task` progress. In this case, since an `Executable Task` keeps track of its current stage, and the workflow is linear, a reset action backtracks one step, provided that a previous stage exists. In either case, the meta information is updated in the storage which causes the UI again to be notified and updated to reflect the changes. This is also the case for the other two actions. As shown on the diagram, the `Task Manager` also relies on these callbacks but that reliance is not fully needed — since it is the `TM` from which the action originates, it can internally update its own state in a more convenient manner.

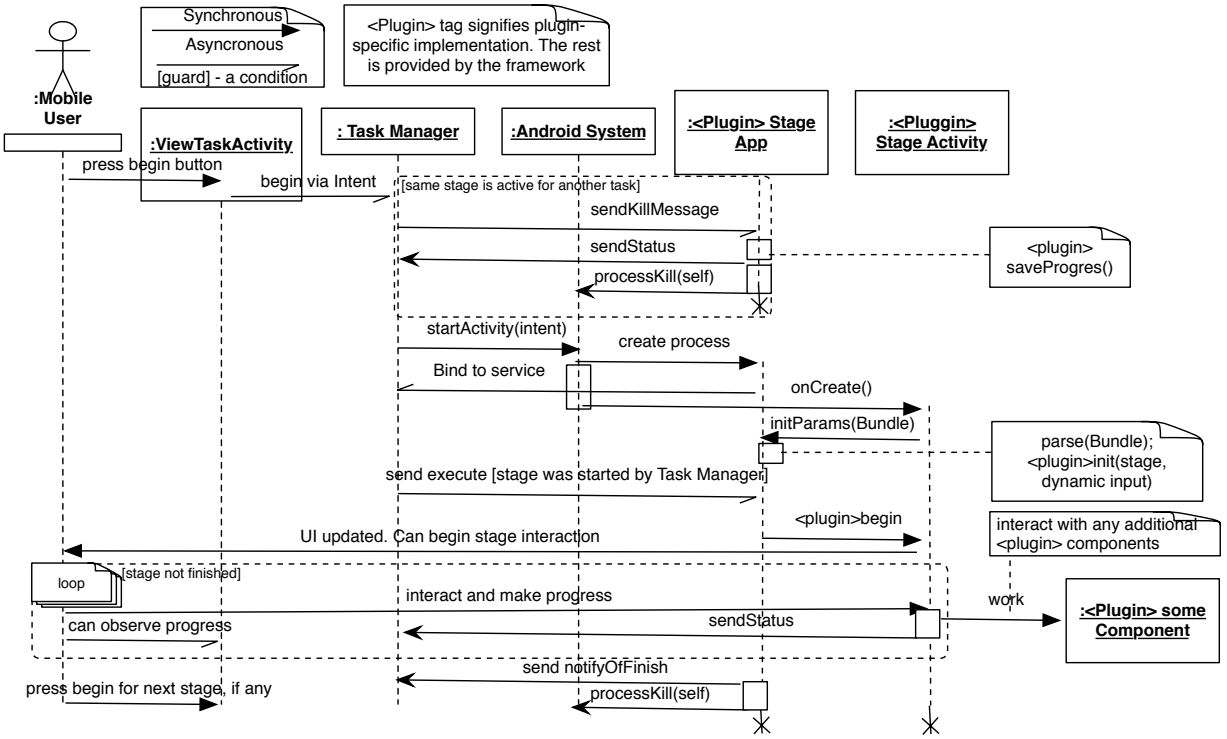

Figure 4.6: Start/Resume a stage plugin.

The most complex of the three actions the `TM` performs is the one used the most often — *beginning* or *resuming* a task. When the `TM` receives a begin *Intent*, the intent must contain the stage id and task id to be started. The `TM` then verifies whether the required stage plugin has been installed and notifies the user if not. It also checks whether the task is one the user has registered for. Most importantly, however, the `Task Manager` checks whether the stage plugin in question is already running, whether because the user was interacting with it up to now (but not any longer), or whether because the plugin has not shut down itself and the Android OS has kept it alive. The second option is possible when the plugin has not finished all the work needed from its previous instance but has been ignored by the user. In such cases, the plugin developer may not have handled the stage's lifecycle properly. Regardless of the reason, if a plugin instance is already running, the `TM` will need to ensure a smooth transition. The `Task Manager` will send the plugin a command to save its progress and shutdown. The plugin will reply its compliance including the latest status, after which it will ask the Android system to immediately kill its own process. After receiving the reply from the plugin, the `TM` proceeds to start a new activity – the plugin's `Main Activity` which corresponds to the unique plugin *type* specified upon registering the plugin with the server. Starting the new activity involves sending all the necessary data over to it. The plugin's `Main Activity`, by library requirement, must be an extension of `Stage Activity`, as shown on Figure 4.8. The data in the intent the `Main Activity` receives is deserialized into a `Stage`

and it is stored in the stage signleton – `<Plugin>StageApp` which extends the `MCSStageApp` the library provides. Any plugin-specific work is done subsequently through the library interface and is controlled by the `MCSStageApp`. During initialization, the `MCSStageApp` binds to the `Task Manager`. More details on the plugin side of the interaction and why the `TM` needs to kill the existing plugin instance is given in §4.2.1.

Once binded (and allowed to exist by the `TM`), the stage application receives a begin command from the `Task Manager` which is then passed to the function `begin()` that the developer needs to implement. From then on, the plugin will respond to status queries from the `TM` in the background without the developer needing to do additional work other than to progress towards the stage's goal, whether via user interaction or through computation.

4.1.4 Notifications, Server Communication & Caching

Push notifications provide a scalable and energy efficient solution to dynamic interaction with the client. The implementation involves registering in the framework app a `Broadcast Receiver` which listens to a particular kind of messages. The implementation leverages Google Cloud Messaging (GCM) service which comprises of a separate server maintaining a channel to the smartphone. This channel is used by all of Google's applications and other third-party applications, too, and it the fastest and most energy efficient way [14] to send push notifications to Android devices. In order to use GCM, the device needs to register with the framework server by providing its unique device id — this is done through a periodic service which notifies the server, if needed. The periodicity ensures the server has up to date information and allows the user to be notified across all active devices. There is a limit of 4KB on the payload size of the notification but this is more than enough for textual input, often sufficient for a JSON representation of structured data, too. If more is needed (for example, for image classification) then the payload can be the URL from which the image is to be fetches. The input type does not matter because it is meant for the specific needs of the stage plugin which knows what to expect.

The work of the `Broadcast Receiver` is to listen for notifications originating from the framework's server, verify their soundness by checking the message contains the needed identification data such as task id and stage id, and create a notification. It can also verify whether the message is still relevant to the user — if the task does not have as current stage the stage referred to in the message, then the notification is no longer needed. The notification is subsequently handled upon pressing it by the `Task Manager` via an intent containing the task and stage ids, together with the payload, and the stage plugin application is started as would have otherwise been started, with the addition of the payload. As shown on Figure 4.7, the notification shows the user the precise stage that will be started and its description but the notification itself is part of the framework, not the stage plugin.

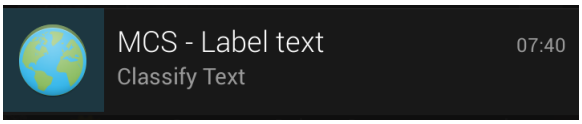

Figure 4.7: Framework notification indicating which is the relevant stage to be activated.

In addition to the push notifications, the client also directly communicates with the framework server. This work is done by the `Communicator` class which is used to register the user and device, to obtain task meta descriptions from the server, and to sign up for tasks. Through followings the Android's recommended way of handling Http connections [35] (i.e, not using Apache Http client but instead using the Android specific client), the `Communicator` is able to set up transparent response compression and fully cached HTTP responses which can completely eliminate the need to communicate with the server when the information in the cache is still valid.

4.2 Stage Plugin Applications

From the very beginning, the goal has been to create a framework application that allows third-party plugins. This is implemented by providing a standalone, self-contained library that acts as an intermediary between the framework and the plugin. This section discusses the provided plugin library and the applications that were built to showcase the framework's flexibility. The varying applications that were built demonstrate that applications with similar usage patterns can also be created. All of the below applications are completely separate applications from Android's point of view. They can be installed and uninstalled separately. There is no knowledge of the framework that was required when building the applications other than what is demanded by the plugin library.

The library implementation ensures that the stage developer does not need to think about managing the state of different `Stage` specifications (instances) that the plugin will be executing at different, possibly interleaved, points in time. The developer only needs to focus on building an application that, when handed in a `Stage` with its parameters, is able to perform the plugin's purpose. The stage application should be thought of a processing unit that produces output for a given user and data input combination. The other work is done by the framework.

4.2.1 Plugin Library & Plugin State

The library consists of four classes and is presented on Figure 4.8 where it is also shown how the plugin application is meant to be used the library.

The `MCSStageApp` needs to be extended by the developer and declared in the `Android Manifest` as the application's `Application` implementation. The `MCSStageApp` provides functionality to the plugin, such as requesting files to write to, data URIs to access, and also ways to let the framework know of its own progress. The `StageActivity` should be extended by the plugin `Activity` that will respond to the unique intent action which is specified upon plugin registration on the server. Extending these two classes requires the implementation of the `ExecutableStage` interface and the `begin()` method. These are the only requirements on the plugin developer regarding the library use. From then on, the developer can do anything with their application. Once the plugin completes its work, it should call `notifyOfFinish()` and everything will be handled by the library.

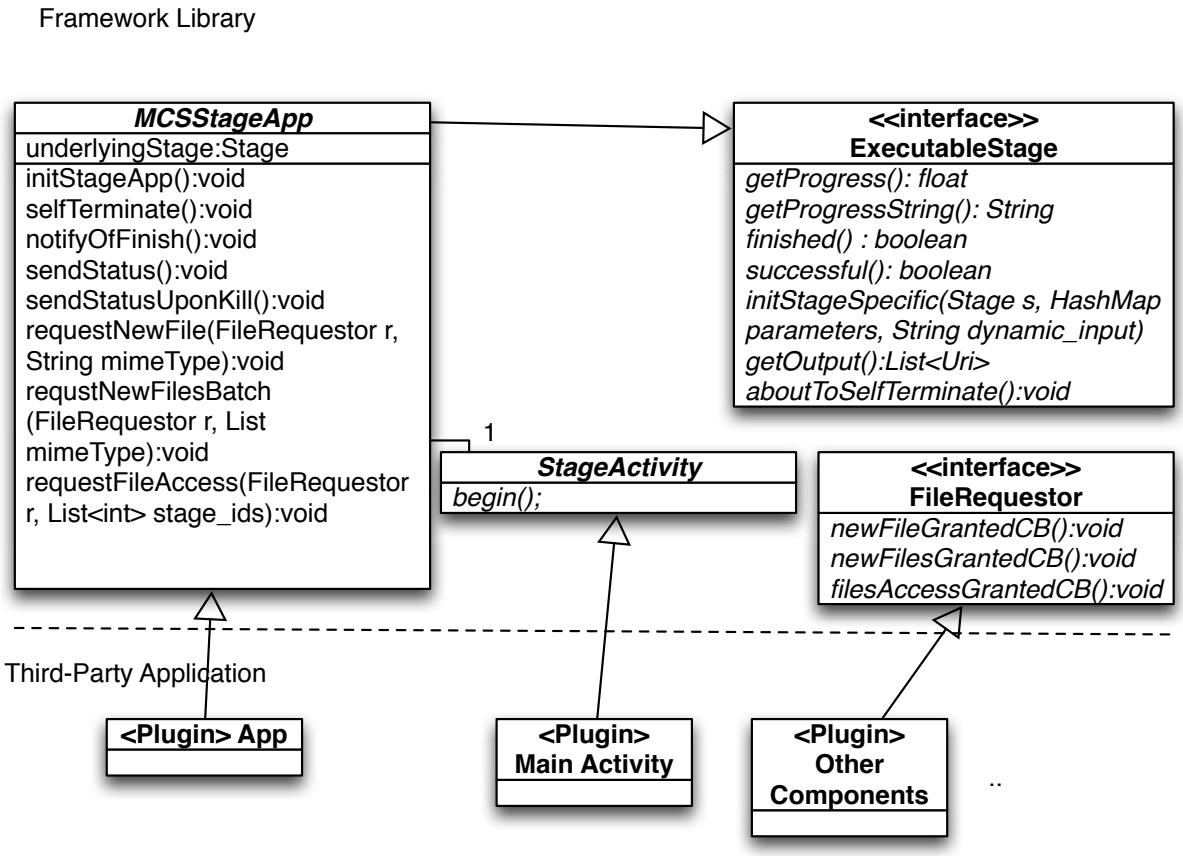

Figure 4.8: Provided plugin library and its relation to third-party applications.

One of the implementation's main challenges has been controlling the lifecycle of the stage plugin. A stage plugin is comprised of the stage application singleton, provided by the framework, together with its extended class and all of the stage application's Android components (comprising the plugin's actual functionality) which are developed by the third-party developer. The library implementation of the singleton class, `MCSStageApp`, is itself is an extension of Android's `Application` class. As is the case for the framework, so does the Android system treat the `Application` class as a singleton. There is only one instance of the class, and it is instantiated whenever any of the application's components is created. It is this realization that the Android `Application` and the designed stage singleton are conceptually on the same level in an application's hierarchy and therefore the stage singleton should extend `Application` that helped solve some of the implementation issues that were had in the beginning. Earlier approaches were to try with using the starting `Activity` to hold the input parameters but this left out stages that were meant to be only a background process without any user interaction. Also, cases where the initiated from the framework `Main Activity` is destroyed if it was no longer used (in cases of a complex multi-screen plugin, or a background-only plugin) meant that either the plugin developers would need to have good knowledge of how to manage the `Main Activity` or that they should do additional work with the received data. A `Service` could have been provided to provide the IPC protocol implementation, but again the developer would need to know how to work with it, and its lifecycle could not be guaranteed which was unacceptable. None of the options above is a realistic option for a third-party developer. Lastly, it is important that the binding to the `Task Manager` happens at the very beginning of the plugin's process creation. This can be ensured only if it is done through the `Application`.

Therefore, the choice to extend the `Application` was a very good decision but it did not solve every problem. There was still the problem of interleaving the execution of stage plugins of the same type. A stage plugin that is instantiated to perform work for one stage cannot be

instantiated for another task without interfering with the first instance. This is because Android's `Application` object is treated (and rightly so) as a singleton by the system — if there are more than one instances of the same component class, they will all share the same `Application` object. Furthermore, having multiple instances of the same component could have complicated matters for the plugin developer who would have then been burdened with ensuring all instances relevant to the same stage input parameters are only interacting with one another but not with components instantiated with other `Stage` parameters. So rather than complicating the singleton object with complex tracking of multi-instance parameters, it was deemed more appropriate to treat the entire plugin application, and each of its components, as singletons. This makes the plugin development much more straightforward.

However, achieving such behaviour had one major complication. The Android system is ultimately the entity that manages the applications' lifecycles. It runs each application in a separate sandbox and treats them as independent. On the other hand, the framework requires that while the plugins are indeed to be separate processes, they should not be independent but instead controlled by it. A conflict between the two can arise when some of the plugin's components signifies to the Android system that it should be restarted if it is killed. Most often a background `Service` would do this. In such case, the Android system will normally oblige and restart the component. In the process of doing this, the plugin `Application` singleton will also be instantiated by the Android system if it did not exist. However, the plugin application is useless without the input provided by the `Task Manager`, i.e. without being started *by* the `TM`. By design, the plugin and all of its components are not meant to do anything else other than perform work for the framework and hence should not exist if not needed by the framework. What such useless existence achieves, however, is to prevent a subsequent 'clean slate' instantiation of the plugin with proper parameters received from the `Task Manager`. This happens because of Android's internal way of starting new Activities — if the `Application` singleton already exists, then there is no need to re-create it, so none of the callbacks that are crucial to the library's control over the plugin will get called. Meanwhile, the singleton will continue to be in an invalid state and the plugin will not be able to operate, the `Task Manager` will not be able to control it (because the callbacks were not executed), thus resulting in a seemingly broken application. Therefore, it is crucial to prevent the Android system from maintaining the `Application` singleton alive for any longer than the `Task Manager` needs it to be. However, it was not known how to prevent the system from doing this and so the next best option was chosen — to ensure that, if the Android system does instantiate the singleton, then it immediately lets the `Task Manager` know of its existence. The `TM` will then check whether the plugin was authorized by the user through the `Task Manager` or not. If it is not meant to be active anymore, then the `TM` will send a kill command, which the plugin singleton will follow and thus ensuring the entire plugin process does not exist. This has been verified in several ways and has been shown to work.

Such additional work would have not been needed if the plugins were to be developed by developers who knew that they should end the plugin's lifecycle immediately after getting the work done, and that there should be no lingering components keeping the singleton alive. However, such a requirement is unrealistic for third-party developers who may allow this to happen for different reasons but most likely it will be unintentionally. This is why the above measures were put into the framework library to prevent this from happening and thus ensure a 'clean slate' for the plugin, regardless of the developer's actions.

Treating plugins and their components as singletons has the limitations that stages of the same type can only execute one at a time. However, this is not a significant problem, since, as it can be observed, a user can interact only with one application at any given time. Plugins that perform only background computation (such as processing data) are the only kind of plugins that can benefit from parallel execution, although it is to be seen what is the likelihood of more than one instance of such plugin is activated at the same time. Background plugins that deal with data production such as sensing, on the other hand, should not be aimed to run in multiple instances for several tasks. Instead it is the generated data that should be made available to all tasks which is in the framework's ultimate goals.

Given the plugin library described, the plugin applications that were built using it are described next. The goal when selecting what plugins to build was to make them widely useful and also to make them different from one another to demonstrate possible use cases.

4.2.2 Sensor Sensing

The `MCSSensorCapture` plugin application collects data from all the sensors specified in Android's `Sensor` class (such as gyroscope, accelerometer, barometer, light sensor). It requires as input parameters the Android constants that are used to denote sensor types. It also requires the frequency with which each sensor type is to be updated (an Android constant, too) and also, how often (in milliseconds) should its value be recorded, and for how long (seconds) should there be a *continuous* recording. These are all specified upon task creation through the web interface at `/create_task` server resource. So is the case with all the parameters for the other plugins discussed next.

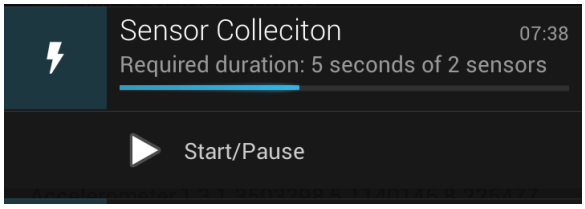

Figure 4.9: Actionable notification of an ongoing sensor collection.

The plugin requests one file per sensor from the framework with the provided file request API. It is there that the readings are stored and later can be used by other stages. It has an optional UI `Activity` that displays the collected data. The UI also allows to start and stop collection but this can also be done through the Notification shown to the user as shown on Figure 4.9. The `MCSSensorCapture` showcases the framework's ability to support plugins that request files to write to and run in the background while having a UI with which the background process is tied with, and a foreground running notification also controlling the background service.

4.2.3 Data Upload

The `MSCUploadData` plugin stage has the purpose of taking the produced data from any other stage and uploading it to the framework's server. The ids of the stages is the required input for the plugin. The plugin also allows to specify an alternative URL to which to perform the upload. The URL must point to a service supporting a multiform POST HTTP request. This is used because the plugin does not know in advance how many output files there are, nor their MIME type — these are established dynamically which provides the flexibility of being able to upload any number of files of any kind. The POST is done over a secure connection and also contains the user information which helps protect their privacy since the data stored on the server is grouped per user. The only permission that the plugin uses is the Internet. It does not even require access to storage. These permissions are visible to the user and make it clear what the plugin does. It is such tight, well-defined purpose that is intended for the framework's plugins. A user can see the description of the plugin and decide if its functionality matches the permissions the application requires.

The plugin runs entirely in the background, only showing a notification of its progress to the user. It is this same progress that is also reported to the main application upon changes, and hence, can be monitored from the framework's application, too.

4.2.4 Text Labeling

The `MCSClassifyText` plugin has the purpose to leverage the user for text classification. The user is asked to provide one of several labels which are specified upon task creation. The other

input parameter for the stage is how many such classifications should an individual user perform before the task is complete. It's main screen is shown on Figure 4.10.

Figure 4.10: Classify text received from the server through a notification.

The differentiating part of this stage application is that it relies solely on dynamic input through server notifications. This makes it much more flexible and it serves as an example of how a mobile crowdsourced solution could be combined with a machine learning algorithm on the server — such notifications can be sent to the user when the algorithm has low certainty; it could be later canceled through the same server API to avoid the user doing work that is no longer required (and furthering their progress). In an analogous way, instead of labeling text, the server can send an image URL which is to be downloaded and shown to the user. This can be used, for example, for the dynamic and interactive assistance for blind people in a similar way to Bigham et al. [3].

The stage's output are all the labels the user has provided. The labels could also additionally be sent directly from the current stage upon the user's confirmation of the label.

4.2.5 Camera Capture

The `MCSCameraCapture` stage is a plugin that can be tasked to either take pictures or videos, and parameterized how many times to do this. The plugin interacts with another application — the built-in camera, to capture photos, which are then saved to a file requested from the framework (in the background with no notice to the user that he has used three, not one application). Once captured, the plugin screen shows a preview of the image captured. Its output are the items the user has captured.

While simple, this plugin showcases how a plugin still has the freedom to work with any other applications over a few screens to achieve its stage goal.

4.3 Server

As stated at the beginning of the project, the majority of work was focused on the mobile client. The work on the server was mostly creating the necessary supporting infrastructure for the mobile client and also laying the groundwork for future work for allowing more sophisticated tasks — ones that allow mixing server execution with client execution, which will enable a more flexible interleaving of machine learning and MCS.

Because the server implementation closely follows the design presented in §3.6 and since the desired cloud service implementation did not involve any major implementation issues that have required special adjustments, this section will not go into much detail about the server implementation. Only a few interesting points will be discussed. It is worth mentioning, however, that just like on the mobile client, there was a plethora of small implementation problems that individually do not pose a challenge but during the development would often arise. This in turn resulted in a large amount of time being spent on such implementation issues. To quickly name a few — library dependency and version incompatibilities, serialization conflicts, non supported libraries by Google App Engine, and more. Such issues are not presented because of their individual insignificance but have contributed to the author's confidence that a framework such as Hive is indeed necessary; less experienced developers would greatly benefit from not having to deal with the implementation issues that the framework solves.

The use of Google App Engine (GAE) was decided as a convenient, out-of-the-box solution to server horizontal scalability. GAE is a Platform-as-a-Service (PaaS) that provides infrastructure and a layer of abstraction from the underlying hardware. It enables the developer to entirely focus on the application without needing to deal with system administration such as sharding a database, reliability, or creating new server instances to handle increased server load. In the context of this project's work, these considerations do not matter but if the framework is to be used broadly as it is ultimately designed, then they will become important. Lastly, having the server be always available and easily kept up-to-date with the current codebase simplified the development-testing-development iteration workflow between the client and server.

Amazon Web Services (AWS) and Heroku were also considered. In retrospect, perhaps AWS would have been a better choice in the long-term because it does not have as much restrictions as GAE does (for example, the kind of databases and file storage one can use are restricted to the ones provided by Google) but would have required more setup work and administrating the infrastructure which would have been of little benefit for the project.

4.3.1 Resources

The server design section (§3.6) already gives a good overview of what each of the functional RESTful resources does and in response to what kind of HTTP request. Therefore there is no for these to be repeated here. Here a few implementation details are discussed.

The `Tasks` resource relies on the `Jackson` serialization library, just as the mobile client does. It is the shared `Task`, `Stage`, and `StageParams` classes that are used but also the server-specific extensions. The former are used only when sending the specifications to the mobile devices while the latter — for the servers internal functionality.

The server's web services are implemented using `Jersey` which is a JAX-RS reference implementation. `Jersey` provides a simplified approach to building RESTful web services. It allows the developer to build normal Java classes which can then be converted into a Resource by providing annotations. Such annotations are the URL path to which the Resource corresponds, the kind of HTTP method to which the Java method responds to, and the parameters (URL path parameters, or content ones) that are injected by the `Jersey` framework.

Not all Resources (as presented in §3.6) allow all four REST operations. Instead, only the ones that have been needed are implemented. For example, some resources either required a PUT or a POST but not both. Nevertheless, each of the Resources has a safe and idempotent implementation of the GET operation allowing the sound usage of caching.

4.3.2 Storage

The data mode described in §3.2 discussed the simple role that the server comes when handling the data. In its current design, it only performs CRUD operations on it, and most importantly, it handles the data by logical entities — whether it is the meta task description or the actually produced data, and hence the decision to use a NoSQL database. In GAE, `Datastore` is the NoSQL solution used for any non-binary storage such as meta information about tasks and participants. On the other hand, `Blobstore` is used for storing the data produced by stages; it is the equivalent of a file system in GAE.

The `BlobDAO` wraps around the `Blobstore` to provide access to a stage data on a per user basis, which is then used by the `Stage Data` resource to serve the data to the same user upon request, or to get all the data from all users which is something only accessible to the `Machine Learning` resource.

The `ParticipantDAO` and `TaskDAO` both use the `Datasore`. Normally, the `Datastore` requires to be accessed through the Java Data Object (JDO) or Java Persistence API (JPA) interface implementations. However, these were deemed too low-level and over-accomplishing for the needs of the server and hence a more abstract external library was used — `Objectify`. Objectify uses annotations on POJO objects to persist them in a typed manner. Later, the persisted objects can be queried by this same type (i.e. class). The `ParticipantDAO` is only a thin layer that enables CRUD operations on a `Participant`. On the other hand, the `TaskDAO` performs plenty of error checking by relying on the `MetaSpecification` resource, to ensure that any task stored in the database is sound. Furthermore, the `TaskDAO` converts `GeneralTasks` — which are the server's representation of the task meta description, and are also meant to contain `GeneralStages` which can run on the server, into their mobile client version. This involves ensuring that the mobile `Task` contains only stages meant to execute on the mobile client. Any stages that are to run on the server are replaced by the generic placeholder stage which can optionally block the task progress until notified by the user.

4.3.3 Task Creation Client

The client is a web page built with a simple combination of HTML and Javascript. Despite its simplicity, it is the essential final step that builds on top of a large number of parts of the framework which enable the specification of a new task without the need to do any development work. Instead, only a couple of parameters need to be specified. The UI is shown on Figure 4.11.

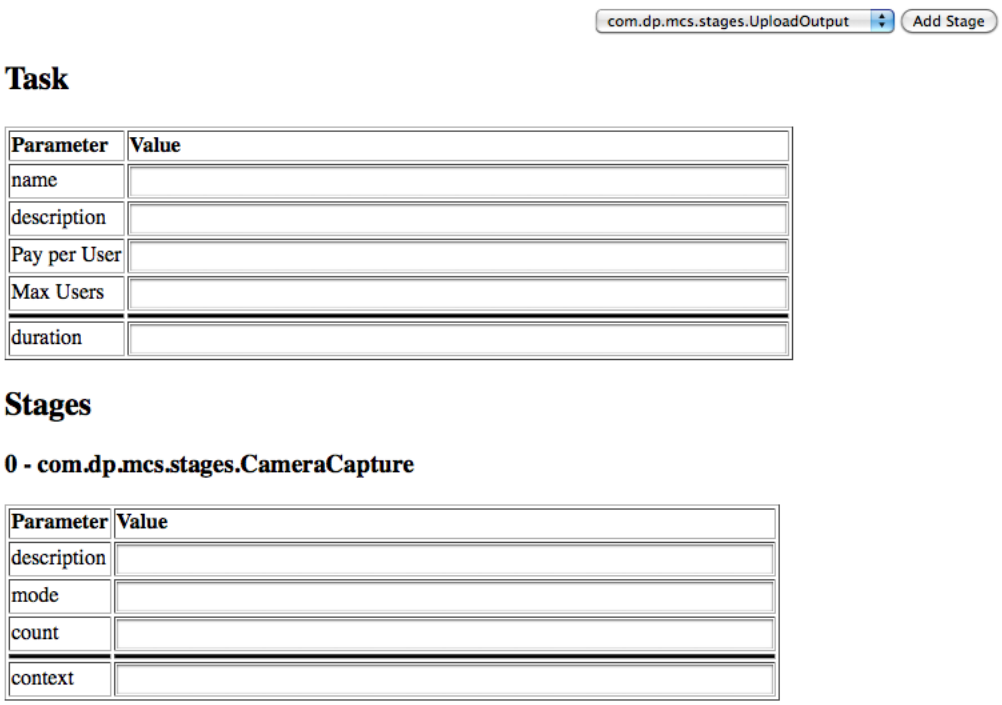

Figure 4.11: Part of the dynamic web interface for task creation.

The UI is much more convenient and successful in guiding the tasker during the task creation process, compared to the Medusa framework where the user needs to specify the task in XML. The web client works as follows. Upon loading the static HTML, the web browser is also instructed to perform a GET to the `Meta Description` resource, obtaining all the necessary and optional parameters for each stage type, and also the overall task. The call is asynchronous. Once done, the UI is updated to present the user with the available stages; the user can keep adding stages to the new task, which on the UI creates an additional table with the needed parameters for that particular stage type. Upon clicking the submit button, the parameters are locally checked against the meta descriptions, and if everything is in order, then a POST request is sent to the `Tasks` resource. Currently the local error checking is only about having filled the required parameters. If there is to be type checking for each one, this would require a more detailed specification of the parameters. The library used for handling the requests is jQuery.

Chapter 5

Evaluation

This chapter discusses the framework resulted from the project's work. Its evaluation is based on the goals (§1.2) and challenges (§2.3) the project set out to address. The evaluation also contains a qualitative comparison with the only other framework identified in §2.1.3 that has similar goals as this project — Medusa [25]. Comparisons to the other framework, PRISM [5], are also made. However, given the larger difference in goals, such comparisons may not always be possible.

During the development and the evaluation of the framework, its correctness was established in various ways and use cases. Different tasks and stages were created, testing for edge cases. Such cases were the interleaved concurrent execution of multiple tasks; the verification that stages of the same type — background or UI related, do not interfere with one another when the user switches between them; the affect of stage resets have on the stage's workflow, data, and soundness; and performing UI work for one task while another was active in the background making progress for the other, to name a few test scenarios. During the tests, extensive use was made of the available Android tools to inspect and debug the execution of the different components, log their behaviour, and also track resource usage.

5.1 Simplified Task Creation

In order to fulfill the main goal of the project — removing the need to design and implement a MCS system for a new task, the system must present a high-level way through which the task workflow is to be specified. Medusa relies on defining a high-level language — the user has to use XML and the predefined XML schema to specify the workflow of a task, the type of stages and their parameters, and what the stage-action transitions should be. However, this solution still requires expert knowledge — the specific syntax of XML and the requirement of how the stages and connectors need to be declared is non-trivial and does require technical background. The user needs to know the names of the executable stages and also precisely what are the parameters they require which is error-prone and burdensome. On the other hand, the presented project — Hive, provides a dynamic web UI (presented in §4.3.3) which wraps around the complexity of specifying a new task correctly. The UI guides the user with the parameters they need to provide, unlike Medusa. In Hive, the tasker is able to specify a new task with only writing the very minimal — only the actual parameter values. In contrast, the Medusa framework requires three times the amount of typing Hive does — in addition to the parameters themselves, the tasker also needs to specify the parameter names — the opening and closing XML tags which are approximately of the same size as the parameter value itself, and any additional XML tags. The Medusa authors report task definitions averaging between 50 and 100 lines of code, compared to only 10-15 lines for Hive of similar task size.

This difference between the two is expected, given the extra layer of encapsulation that the project has; nevertheless it is important for the end user and precisely why should be implemented. For both frameworks, the minimal amount of task specification might be all the work needed to create a new MCS task without any implementation whatsoever. However, if parts of the task have no equivalent plugins available, these will need to be developed, in addition to the task specification.

Such development will be fully needed when building an ad-hoc application, too. The benefit of the framework over ad-hoc application is in providing a solution to any of the parts of the task for which there is a plugin.

Despite being more verbose than Hive and requiring technical knowledge, Medusa, too, is deemed highly efficient, compared to the normal ad-hoc mobile crowdsourcing development process. In the case of ad-hoc applications, there is no task specification part but only the actual application implementation in the programming language of choice. These standalone implementations are likely to measure in the thousands of lines of code; in fact, in their evaluation, Ra et al. [25] report that Medusa is two orders of magnitude shorter than the compared corresponding standalone application. Such a vast difference in the development work is the driving motivation for frameworks like Hive and Medusa.

PRISM [5] is an interesting case. It too, like ad-hoc solutions, does not have an extra abstraction for a task description; it only relies on the executable. However, in the very unlikely case where the entire task is equivalent to something which already exists and it is setup with precisely the parameters needed, then there is no development and no task specification to be done whatsoever. Only the binary is to be supplied to the system which will distribute it to clients. In other words, its the most succinct way possible. However, it is highly improbable for this to happen given that PRISM is not designed for combining binaries to achieve a single task, unlike Medusa and Hive but instead is meant to be a execution engine that can run any binary. PRISM relies that the taskers will find ways to combine existing libraries on their own. This puts a much bigger technical burden on the tasker, compared to Hive. Therefore, in the the realistic case, the PRISM users will need to develop their own solution which in size will be of the same order as ad-hoc solutions.

Despite meeting one of the primary project goals, Hive's task creation process can be improved in one or more ways. First of all, the UI should provide a richer guidance to the user of the type of the parameter they should provide — integer, a collection of integers, a string. Currently this is not done but instead it is relied that this information is being inferred by the appropriately named parameters and that the tasker has had some instruction of how to use the system. Such additional information presentation requires that the underlying declarations of the plugins become more explicit and provide type information about each parameter and possibly — a range of values. However, this can make the plugin registration more cumbersome and hence needs to be addressed in such a way that both the plugin developers, and the taskers benefit.

In addition, the UI should provide tools to edit tasks and track their progress, the number of participants, and also provide access to the implemented notification service in a more convenient way than the current (which is knowing the precise URL to which to send the payload). These are mostly additions that deal with the presentation to the user, not the underlying server workings so they should not pose a challenge but only require time to implement.

5.2 Generality

The four plugins that were provided (§4.2) can be combined in any permutation to form a task of arbitrary number of stages. The plugins were selected and developed as such to showcase the support Hive gives to plugins and to act as an example to new plugins. Each one of them performs differently from a application behaviour point of view — one interacts with other applications, another works completely in the background. The third plugin presents both a UI and a notification through which to interact with a background service, while the forth relies on dynamic input from the server to operate. These different aspects of theirs can be, of course, all leveraged in a single plugin. What's important is that at least one of these patterns will occur in almost any kind of stage plugin which means that such plugins can be built in a similar fashion and the four plugins. There is no restriction on the kind of APIs that can be used — a plugin has the full power of a standard Android application. This means that any of the data post-processing approaches discussed in the literature review can be implemented as plugins; these are important for more complex task workflows.

Given the freedom on the plugin level, what is left to consider is the flexibility of specifying tasks. Hive is slightly less flexible than Medusa in terms of the kind of tasks that can be specified; this is due to the sequential workflow that is only supported by the framework while Medusa also allows a fork-join construct. For example, it will not be possible to specify and execute tasks that concurrently run stages on the client and also on the server, or run concurrently more than one plugin on the client. The second issue, as as pointed out in the workflow section (§3.3), is not significant and can be easily circumvented by creating a plugin which internally achieves the concurrency. Nevertheless, it may involve some functionality duplication. However, when it comes to concurrent client and server stage execution, this is indeed a limitation. Similar to Hive, Medusa does not support such a task, either. In fact, Medusa's goal is the execution of stages on the client; there is no design involved for the server to perform any work specified by the task. In contrast, this project has presented a very modular server design that has independent components with precise and simple external APIs (REST). The server is implemented to allow the orchestration of the components through these APIs. What is currently lacking is the engine itself but it may not even be needed to be part of the framework, given the simple APIs. For example YAML or JOpera could be used to orchestrate the RESTful web services on the server externally.

In principle, PRISM should be able to achieve what Medusa and Hive cannot — concurrent client and server workflow, given the complete binary freedom but it must be stressed that it is up for the task requestor, not the framework, to create such workflow. PRISM itself does not control it; it is the task binary should contain the execution control. Such design strongly conflicts with the project's goal of non-programmatic task creation and code reusability.

Given the use cases of MCS found in the literature (§2.1.2), this paragraph discusses if they can be implemented and how in the Hive framework.

VizWiz [3] provides an interactive dialogue between blind people and the crowd which offers navigational and recognition assistance. The Hive realization of this use case can be a task representing a user needing assistance. Users who sign up for the task signify their readiness to help. When a blind person takes a photo and uploads it to the server (which can be another task, and both tasks be linked through a server stage plugin which leverages the notification service), a notification is sent to the participants who can then activate the 'assistance' plugin stage that is part of the task and respond. Upon the pre-agreed number of assists or their own desire, the participants can end their work and claim any rewards. The final stage, returning the collected crowd input to the user is done through another notification, this time sent to the person requesting the help. The notification can trigger a stage plugin meeting the special needs of VizWiz — guiding with sound the blind user of an objects location or the other options discussed in [3].

To achieve the task, several plugins need to be used. The camera capture and upload plugins are already available, while the text classification plugin can be used as a template for creating the photo assistance plugin — instead of text, display an image to provide textual feedback of. It is only the server stage plugin of tying together the output from the upload plugin (i.e., data stored on the server) as an input to the follow up stage of using the server notification service that needs to be implemented. It is fully supported, however. Legion [18], the crowd-curated software UI control can be implemented in a very similar task structure the task to replicate VizWiz.

Mohan et al. [23] present a MCS application for detecting traffic conditions and pot holes along the road. The data collection plugin itself is already available. What such a task should include is additionally the server plugin stage that analyzes the data. Such stage can be based on the machine learning service, presented in §3.6.2. However the service itself is not implemented so currently the full application is not possible — only its data collection.

In summary, while the chosen workflow does provide some limitations, these can be worked around to implement the MCS tasks discussed in the literature review. Tasks can be specified in a high-level way with no implementation needed if there is an available plugin. The Hive framework requires less technical knowledge and is less input from the tasker, compared to Medusa, which is already successful at reducing the burden of creating new mobile crowdsourcing tasks. The mobile

client supports any kind of processing plugins and its usability can fast grow, given the decentralized extensibility discussed next. The key drawback of the Hive framework is that in its current form does not provide execution of stages on the server. However, this is an implementation matter only because on a design level it has been made possible, given the presented web services. The resolution of this drawback is key to enabling server stages that rely on machine learning and hence the long-term goal of interleaving ML with MCS.

5.3 Decentralized Extensibility

The delivered framework has been designed from the very beginning with third-party developers in mind. Hive provides a library that guides the development and interaction between the framework application and third-party plugins. This library is self-contained and can be deployed during the development of any plugin; there is no need for the developer to have any access to the source code of the main application nor to understand how it functions but only to know where in the conceptual design does their plugin and the framework fit together. Also, it is up to the developer how to distribute the plugin; as long as it is registered with the framework, it can be used in tasks. These form a distinguishing feature which cannot be made possible without certain design decisions. Such decisions are the creation of the plugin library, the management of the plugin lifecycle Hive does, and also the full control over the user's data. Such decentralized extensibility is not present on Medusa where only the framework's developers and anyone else who has full understanding of the system and access to the framework's entire source code can provide new plugins. It is one of the main differences between the two frameworks.

The library requirements are straightforward. Each of the library's public methods have well-documented source code and error checking to ensure the library is understood by the developer. There are nine methods that are absolutely necessary to be implemented by any plugin; there are an additional two methods if the plugin needs to write data and access data produced by others. Seven out of these eleven interface methods, however, are quite trivial. In all of the four plugins that were developed, these seven methods (which perform actions such as returning information about plugin status and progress, and getting the plugin's output URI) were all implemented by a maximum of three lines of code but on most places, one line was sufficient. Even more, the implementations of these seven methods between the different plugins were very similar if not identical — for example, each of the stages presents the progress as the fraction of completed items over the total number of required items.

This observation also suggests, however, that such functionality could have been handled by the framework, given the commonality between the plugins. Yet, not every plugin might have the same kind of progressing as the ones implemented. For example, there may be stages that have no precise finish point but instead it is up to the user to decide when they would like to stop and move on to the next stage. Such plugins will need to present the progress and handle their status in a different way to the developed four plugins. Nevertheless, the ideal solution would have been to decouple the plugin status and progress reporting from the rest of the library and present the available options of reporting progress to the plugin developer — either through the plugin library, or upon plugin registration with the cloud service. However, this is quite a simple issue, and given the minimal development effort these methods need, perhaps the proposed change is an overengineered one.

The remaining four functions that will usually be needed to be implemented are more involved. Yet, these methods are a small part of what needs to be implemented for the overall functionality of the plugin. For the four plugins developed for the framework — multiform HTTP upload, capturing photos or video, capturing sensors, labeling dynamically provided text, all the library implementations (that is the entire `StageApp` class implementation) are between 70 and 90 lines of code. In fact, all these classes have only one method which is longer than 5 lines of code — that is the `initStageSpecific()` method which receives the `Stage` description and any input parameters from the framework application which are handled according to the plugin's needs. This method is on average 20 lines of code.

The important thing to note is that many plugins are likely to have similar sort of library implementation size. As the complexity of the actual work the plugin increases, the size of the part conforming to the library interface will be more and more negligible in the overall development size and time spent, at the added benefit of the plugin being reusable by anyone using the framework. As an illustration, three out of four of the developed plugins have in total (`StageApp` implementation included) of 300 LoC each while the sensor capture plugin has 600 lines of code.

A testimony to the simplicity of the plugin library requirements is the fact that the last two plugins which were developed as part of the project — the text classification and stage upload were developed fairly fast. Given the already gained Android experience during the project work, it took the author only 3 days to develop both plugins. Granted, this would have taken more time to someone who has not worked with the library. However, the development process would have taken significantly longer if in addition to writing the core plugin functionality, the author had to implement the following features as part of the 2 applications: a server with a web service to send notifications leveraging GCM, a web service to receive multiform HTTP upload, a hosted storage for the uploads, an SQL database on the mobile phone to track progress between application lifecycles and different tasks, a server database tracking user sign ups and participation in the application, and a way to access the uploaded data.

5.4 Data Privacy

The framework encourages the development of plugins that have a single purpose which makes it easier for the end-user to understand the plugin's capabilities. For example, the plugins cannot write to the external storage (and thus potentially expose sensitive data to uncontrolled access from any application) unless it declares this in its own Android manifest. The user can leverage this information to decide whether to install the plugin or not, as it is done with any other application on the Android Market. More importantly, however, is that Hive has total control over the private data generated by any of the third-party plugins. Conversely, none of the plugins has any access to it or a permission write to the framework's private storage unless the user allows it when asked by the `Task Manager`. Through the `TM`, the user has complete control over their data which is at the very core of their privacy.

To a similar conclusion have reached the Medusa authors who perform static code analysis to detect any stages which access the Internet or the file storage in which case these stages are rejected. In this way they are able to ensure that the user's data stays on the mobile device and is never moved from Medusa's application. However, their approach also means restricting that no plugin can access the Internet which affects the kind of tasks the framework supports.

However, Medusa is less obtrusive to the user than Hive because Medusa will ask the user's consent only when uploading data to the server unlike Hive which does this upon any data access which can be annoying to the user. Hence, Hive can be improved by providing a more sophisticated control over the data. For example, remembering previous permissions over a period of time or per task, and allowing the user to set preferences regarding their privacy would be really useful. Fine-grained control over the device's sensors would also be beneficial, not only for the user's privacy but also battery drain — for example, only allowing a subset of sensors to be active. Such functionality is present in PRISM but not in Medusa. However, limiting sensors can clash with task requirements, to there needs be a clear way to let the user know about the trade off between participating in a task and their privacy settings. Lastly, as suggested by [8], data perturbation is a post-processing way to make sensor data non-identifying. As shown in the previous section about Generality, such type of plugins is certainly possible to create and use in the framework.

5.5 Performance

This section will briefly discuss the performance on the mobile client. The performance of the server cloud service has not been investigated because of its lightweight purpose which makes the

evaluation mostly an evaluation of the underlying infrastructure.

The Android Dalvik Debug Monitor Server (DDMS) tool was used when profiling the performance of the application. It provides a holistic view of all the processes running on the mobile device and allows to time and view the CPU use of each method executed in the system.

The aspects that were profiled were the device registration with the server, the notification service, downloading the server's available tasks, and the communication between the framework application and the plugin. These were deemed to have the biggest impact on the user experience and the application performance.

The device registration is a service that executes from time to time (whenever the underlying GCM system channel that is being leveraged changes the id assigned to the device). The implementation follows the recommended way [14] of checking when to update the server with the device's id. The device registration was profiled to awake and send the details to the server and then completely close until the next time; it does not linger in the background. This process itself takes on average 350ms which is roughly the request-response average latency for the server given the minimal payload that is sent.

More interesting is its counterpart — the Notification Service. The entire time to process a notification — from the moment it is received by the Android system to the moment the user is shown the notification (which involves querying the database to obtain the stage's details to include in the notification), is always between 35 and 45 milliseconds. In conjunction with the framework's server average latency of 300 ms and assuming (and having observed in every test during development) similar response and handling latency by Google's GCM servers, the overall cumulative time is still less than a second! In the author's view, of all the performance measurements, both for this report and Medusa's, the notification delay has one of the biggest overall impacts on the framework. The delay is not only a performance measure but a measure towards the framework's generality — the low notification delay acts as an enabler for tasks that require near to real time responsiveness. For example, for MCS applications such as the the interactive help for the blind [3], as shown in the literature review. Contrast this delay with Medusa's delay which is reported in the presented paper to be between 17 to 78 seconds. An application cannot be called interactive with such a delay. The delay is because the authors rely on email-to-SMS gatewaying and AMT to distribute the notifications and therefore such delay is to be expected. Even if Medusa is made to reuse the same underlying Android channel as Hive is, it is still unlikely its notifications will reach the 35-45ms latency on the mobile device as Hive does. This is due to the design choices made for Medusa — the mobile client does not have the entire task description; instead, every stage is sent via a notification only when its time has come. This, however, makes the notification much more cumbersome — both in size and computation, compared to Hive's. In Hive, the notification only has two parts of information. The absolute minimal task identifying information — task and stage ids, and the optional payload for dynamic input.

The next part of the system that is viewed to have a big impact on the system and hence profiled is the `Task Manager`. Its main purpose is to act as a mediator, and through this mediation — to perform plugin orchestration. Therefore, it is not the `Task Manager` alone that needs to be profiled but its interactions with the plugins. The most frequent action the `TM` does is starting stages. What was profiled is the time to prepare the data for a new stage, send it, and have the stage bind to the `Task Manager`. On the test device — a Samsung Galaxy S3, it takes 200 ms for this to happen. It takes the `SensorCapture` plugin, the most complex of the four provided, another 200-300ms to initialize itself, send a ready message to the `TM`, and receive a `begin` command. This time involves plugin-specific initialization but also deserializing the received `Stage` description and parameters (recall that the framework and plugin applications are separate processes and hence any data exchange happens through messaging). The deserialization takes longer than compared to the same action on the `TM` because it is done by a new `Jackson` object.

The `TM`, on the other hand, uses a `Jackson` object that is shared together with every other element of the framework application. This was an implementation decision. It allows `Jackson` to cache its internal workings which enables a great speedup. During the benchmarks performed,

the fresh Jackson object would take close to 100ms to write a Task into to a JSON string while after its first run, this time would be reduced to 3-4 milliseconds. It is similar with creating a Java object from a JSON string, although it does take slightly longer than the former action because a more complex object is created in the VM. Such a large difference in time between fresh and cached serialization stresses the importance of having searched for a fast JSON serialization library such as Jackson because other libraries may be less capable of such runtime amortization. A slowdown could have been felt when there are multiple concurrent stages. As seen on Table 5.1, Jackson's efficiency also benefits the time to communicate with the server.

In addition to the begin action, the TM actively interacts with the plugins for status updates. This is the most common interaction between the two sides which is why it was made by design to be lightweight. The status exchange contains only primitive data types that need not require serialization and deserialization. When profiled, it was found out that handling a status update takes on average 25 ms, including the database update. The other types of messages, such as file access request and file creation request take slightly less — 18-20ms. If there are many active plugins, the updates can be completely parallelized per plugin as they do not affect one another.

Lastly, the performance of the client-server communication when retrieving the available tasks is measured. This is directly visible to the user, unlike for example, the TM-plugin interactions, which is why it is useful to optimize it. What was profiled was the time it takes to communicate with the server, deserialize the received data, store it and display it to the user. Only the last part is done on the main thread (i.e. UI thread) which affects the user's perception of responsiveness. The results are summarized in Table 5.1. The time displayed was averaged over 5 runs.

Method	Initial	Cached Response
send and receive GET	1050	450
JSON to Object	550 (cold start)	70
store tasks	75	12 (skipped)
load tasks to UI	60	0 ms (skipped)

Table 5.1: Refreshing 4 fully specified tasks from the server. Time is averaged over 5 trials, in msec

Recall that by following Android implementation guidelines [35], the mobile client was enabled to rely on cached server responses. These can be seen in the table. As it can be seen, the cache reduces in half the time to browse the four tasks that were set up on the server. As the number of tasks grows, this benefit will grow even more even if only subsets of the available tasks are retrieved. This is because the 450 ms of the Cached Response are only the wait time to receive a 'nothing changed' response from the server, i.e. response does not contain the task data. Finally, the size of the data itself was measured. The JSON generated for the four tasks was 4.5 KB. On the other hand, the XML representing the same tasks is 10.5KB. As discussed in previous chapters, this difference matters, given the large-scale goals of a crowdsourcing framework.

In conclusion, the implementation choices — selecting Jackson, following Android principles for building the notification and communication services, and having a lightweight library for control over the plugins, have enabled the framework to achieve its functionality with minimum impact on the user experience and the phone battery — the achieved fast running times of actions such as status updates and serialization are the result of a low CPU needs. This in turn has an impact on the battery. On the other and, IO bound actions such as the refresh from the server or receiving notification, are important for the application's perceived usability as they are directly observable by the end user.

5.6 User Experience

The user experience comprises of the direct interaction with the application and the indirect impact the framework application has on the mobile device. The indirect influence is directly linked to the

application's performance which was already assessed. To add, only the `Task Manager` is active during the entire lifespan on the framework application; however, it performs no work if there are no plugins active and hence has minimal impact over the CPU.

As shown in the previous section, the application is fairly fast in its operations which allows for fluent UI. During the various test scenarios that were described in the beginning of the chapter, the UI was always kept up to date with progress information about all the tasks, no matter the screen transitions that were tried. This showed the good decoupling that existed between the view and model implementations in the application. The UI also allows viewing available tasks directly from the application which is a better option, compared to Medusa where the user is asked to browse and sign up for tasks through visiting AMT's website using the browser. Medusa's solution takes much longer (loading additional HTML and images for the web page) than refreshing from the server shown in the last section.

However, the user experience can be much improved. The UI should provide much richer information about the stages and also about the current tasks that the user has signed up for. Sifting tasks using various filters such as the type of stage plugins used, sensor usage, or current contextual applicability will be quite beneficial.

5.7 Summary

The newly proposed framework meets its main goal of making it very easy for a non-expert to create a new MCS task without requiring any development, or in the worst case, the development effort is reduced. The evaluation has shown that the requirements such a goal sets were overall met. In several key aspects, such as support for third-party developers, scalability, and task creation guidance, Hive surpasses the only other framework that provides such functionality — Medusa. However, there are aspects in which Medusa is better (such as task creation verification) and additional shortcomings that should be addressed. The comparison between the two frameworks is summarized in Table 5.2.

Item	Medusa	Hive
Task Creation		
Reuse plugins	Yes	Yes
No programming needed if possible	Yes	Yes
Error checking	Yes(level unknown)	Basic
Non-technical entry	No	Yes
Minimal Typing	No	Yes
Tracks task progress and other tools	Unknown	No
Generality		
Fork-join construct	Yes	Only via plugin
Limited APIs for plugins	Yes – no storage or Internet use	No
Execute stage on server	No	Designed; groundwork implementation
Restricted by server connectivity	Yes	No
Postprocess data	Yes	Yes
Decentralized Extensibility		
Extensible	Yes	Yes
Requires full core framework understanding	Yes	No
Requires access to framework source code	Yes	No
Provides plugin library adapter	No	Yes
Data Privacy		
User in control	Yes	Yes
Plugins can write to disk	No(static code analysis)	Up to user
Plugins can use connectivity	No(static code analysis)	Yes
Control access to private data	No	Yes
Plugins have access to private data by default	No	Yes
Minimal attention needed from user	Yes	No
Performance		
Real-time notifications	No	Yes
Client-Server Comm	Approx. 1 second	Sub-second
Scalable	No(§2.1.3)	Yes
Lightweight serialization	No	Yes
Communication Caching	Unknown	Yes
User Experience		
UI always up-to-date	Unknown	Yes
Robust and responsive UI	Unknown	Yes
Integrated task browsing and information viewing	No(visit AMT page)	Yes

Table 5.2: Summarized comparison between Hive and Medusa according to the project's goals.

Chapter 6

Conclusion

6.1 Project Outcome

This project set out to ameliorate, completely or at least substantially, the need to design and implement a mobile crowdsourcing system — something requiring both a server and a mobile application, whenever a new way to leverage the crowd is thought of, or someone needs to simply collect and process data from the crowd. The goal was motivated by the fact that often the development of such a system consumes a lot of time and effort. The author of this report was convinced of this first-hand while working on the presented framework — in addition to the discussed design and implementation challenges, there were other smaller implementation issues that on their own are still non-trivial to overcome but nevertheless not worthy of a mention due to their obvious solution. However, when accumulated and combined with the larger challenges, these issues become a major roadblock to realizing the system's purpose and only add to the serious time and development commitment needed to implement the entire system. Such effort feels unjustified if the MCS system itself is not of much interest but is only seen as a prerequisite to the evaluation of a MCS application, the training of a ML algorithm, or simply getting input of the crowd for non-technical purposes.

As the evaluation has shown, the project's main goal has been accomplished — a new MCS system need not be implemented for the above purposes. The presented framework, Hive, is a solution which can be leveraged for performing various MCS tasks with no or substantially reduced development effort. It is comprised of a mobile client and a server, both of which provide common functionality that can be reused for new MCS tasks, removing the need to deal with many design and implementation issues. Some of these services are managing user participation, empowering the user to control their data privacy, and the actual orchestration of task execution. Hive offers a high-level way to declare the workflow of a task and requires an additional implementation only if there is no plugin satisfying a given part of the task's requirements. The new framework is much more succinct in the task specification than Medusa which is already quite efficient when compared with MCS ad-hoc solutions which are actual source code. Hive also requires less technical knowledge than Medusa because it hides away the complexity of task representation; this makes it possible for it to be used by non-technical people.

There are other aspects in which Hive stands out; such is its practical extensibility. When Hive's plugin collection is insufficient and new plugins are needed, they can be easily developed and contributed to the framework. What makes Hive unique is its orientation towards third-party developers. This has been achieved by ensuring that developing for the framework is straightforward, does not impose a large burden on the developer, and does not require knowledge of the framework's internal structure and implementation. The design of the plugin library is what makes this possible. It acts as an adapter that hides away all the framework's complexity and also performs some 'heavy-lifting' work in the background, so that on the surface it is a simple API that can be leveraged by the plugin to fit into the framework. The conformance to the library's interface, as suggested by the results obtained in the Evaluation chapter, comprises a small part of the overall plugin implementation and it is unlikely to change in size, whether it is part of a simple plugin or

a very complex one. No other solution in the literature has aimed to meet the goal of extensibility through such decentralized, third-party plugin support while still offering a non-expert way to specify tasks by reusing existing functionality. The only other two MCS frameworks do one of the two but not both — PRISM allows for any code execution but there is no solution reuse or, in fact, any task specification, only implementation. Medusa, on the other hand, does not offer any help to third-party developers — they must know the entire system in order to contribute a plugin to it. Yet, decentralized extensibility is the most realistic and practical way to ensure the framework's growing generality and hence, its future use. The alternative — a centralized-only contribution, is a bottleneck to the framework's extensibility and speed of development.

As the Design and Implementation chapters have shown, making the framework 'friendly' to third-party developers was not a trivial feature to achieve — an additional view had to be taken on data privacy and how it can be safely shared between plugins, a working implementation for a non-typical Android application — one controlling others which are not implemented by the same developer, had to be built, and performance and interference between plugins of the same type had to considered. The resulting framework application is able to exert total control over the plugins' lifecycles, to orchestrate them for its own purposes (task execution), and ensure restricted access to private data, while at the same time not limiting their functional freedom nor the flexibility with which the plugins can be combined and their output data — used. In the context of the present popular smartphone operating systems and to the best of the author's knowledge, such solution has not been presented before. Its design and functionality is not specific to the subject of MCS and therefore can be adapted to other contexts and projects which require a mobile application behaving like a framework and necessitating similar relationship with third-party plugins.

Finally, Hive solves some of the challenges unique for MCS which means that instead of such solutions missing from an ad-hoc MCS application, as it can often happen, they are offered to Hive's users as a ready implementation. The framework has a strong focus on preserving the user's privacy. Furthermore, as shown in the performance section of the Evaluation chapter, the mobile client is a robust and fast application. Combined with the design choices about the server, and also the decided separation between the client's and the server's responsibilities such as the mobile clients self-managing the task execution and the server not needing to track the entire state of each participant for each task, these characteristics collectively render Hive as a more scalable solution than Medusa.

Nevertheless, not all of the project's goals and requirements were fully completed. In addition, this project can act as the starting point for a few future projects. These are discussed next.

6.2 Future Work

The Evaluation chapter has pointed to some of the unfinished implementation work and improvements which can be made to increase the usefulness and capabilities of the presented framework. For example, there is a need for a more sophisticated guidance of the task creation process — the user should not have to infer the type (integer, list, and so on) of the parameters from their names. These should be part of the framework and error checking performed during task entry. Furthermore, the tasker should be presented with a set tools to view information about the ongoing tasks they have specified, and the framework's collection of plugins should be expanded with new plugins that solve additional common problems. Such additions can be, for example, classifying images, performing local analytics, or pointing locations of specific interest on a map. Enhancing the mobile client's capabilities will be quite useful, too.

Also, it will be much beneficial to the mobile user and it will lead to improved task participation if the users were provided with filtering functionality with which to sift through the available tasks. Furthermore, the UI should present more comprehensive information about plugins, their capabilities and the input parameters that they are set up with for the given task. Lastly, the machine learning service needs to be completed. While the experiments undertaken with Weka have led to some preliminary results showing that the implementation will certainly work, the service is not fully completed and hence the framework currently cannot support tasks which can

involve the training and usage of a ML algorithm.

However, these are additions are mostly implementation-related enhancements and their main requirement is the time needed to add them. There are, however, a few major directions that future works could explore.

Of great importance is completing the mobile context service solution on the mobile client which design was discussed in §3.5.4 but due to time constraints was not possible to make fully operational. The `Context Service` will improve the user experience in finding suitable tasks because when combined with the filters, the user will be able to look for tasks that are specifically applicable to their current context. Such 'niche' tasks could be of higher reward, given that not everyone will be able to participate which will increase the importance of each potential participant to the tasker. In fact, the very existence of such tasks will be made possible — without a guarantee than the data collected is from a specific location, time, activity, or some combination of these, it is likely that the data produced will be very poor and no tasker would trust the framework with a task with such requirements. Equally important, the context service will have a positive impact on the accuracy of any data collected — the mobile context could be used as extra information that will help to better assess the circumstances in which the data was collected and thus gauge its trustworthiness.

The use of mobile context as a data attribute will be a stepping stone to addressing the second main shortcoming of the framework — data reuse. While access and reuse to the framework's data is currently possible, it is only through explicit pointers between stages. This means that in order to reuse the data, the tasker must know of the existence of the containing task and stage; this makes the current data reuse possible only in a very limited scenarios. Of great benefit would be the addition of an automatic service which is able to determine that the data produced from one stage instance of one particular type can also be treated as the output from another stage instance of the same type. An even more advanced solution would be the detection of data that can be reused for another stage instance but of different type. This will require comparing not only the initializing parameters of the stage instance as the first solution will, but also an analysis of the type of sensors used to produce data as well as the function calls the plugin does — i.e., the use of code analysis techniques. Such solutions can easily be a project on their own and hence it was decided that it was not possible to fit them within this project's time frame.

The server's capabilities can also be the focus of future work. First of all, it should be experimented and decided which option is better — having an external workflow execution engine such as YAML, BPEL, or JOpera, or developing one internally, as it was the case for the mobile client. This decision might result in some design choices — it is not expected that any of the existing parts will change given their independence but there might be new parts added to the system that will need to be integrated in a way that respects the framework's goals. For example, an external engine will require translating the task specification (or extending it) to the needs of the engine while an internal one will mean designing a module with goals and considerations similar to the mobile client's `Task Manager`.

Once the choice of how to implement the execution module is done, the framework will be able to support tasks that specify execution both on the server and client. This will enable the integration of ML into MCS tasks.

Appendix A

Plugin Implementations of Library Requirements

Listing A.1: Upload Plugin's implementation of the framework requirements

```
import com.dp.mcs.shared.bean.Stage;

public class UploadApp extends MCSStageApp {

    public static final String STAGE_ID = "stage_ids";
    public static final String ALTERNATIVE_URL = "alternative_url";
    private static final String DEFAULT_URL = "https://mobilecrowdsrc.appspot.com/api/tasks/";

    List<Uri> files;
    ArrayList<Integer> stage_ids;

    int filePtr;
    boolean finished;
    String url;
    protected boolean successful;

    String fileUploadName;

    @Override
    public String getProgress() {
        return String.format("%d/%d", filePtr, files.size());
    }

    @Override
    public float getProgressFloat() {
        return ((float) filePtr)/files.size();
    }

    @Override
    public boolean finished() {
        return finished;
    }

    @Override
    public boolean successful() {
        return successful;
    }

    @Override
    public void initStageSpecific(Stage s, HashMap<String, Object> params, String dyn_input) {
        files = new ArrayList<Uri>();
        stage_ids = new ArrayList<Integer>();

        String[] stages = ((String)params.get(STAGE_ID)).split("\\s*,\\s*");
        for(String str:stages){
```

```
46            stage_ids.add(Integer.valueOf(str));
          }
          if(params.containsKey(ALTERNATIVE_URL)
                && ((String) params.get(ALTERNATIVE_URL)).trim().length() > 0){
             url = (String) params.get(ALTERNATIVE_URL);
51        } else {
             url = DEFAULT_URL + String.format("%d/%d",
                                    getUnderlyingStage().getTask_id(),
                                    getUnderlyingStage().getStage_id());
          }
56        fileUploadName = String.format("task%dUploadStage%dUser-%s",
                                    getUnderlyingStage().getTask_id(),
                                    getUnderlyingStage().getStage_id(),
                                    getUserName());
       }
61
       @Override
       public ArrayList<Uri> getOutput() {
          return null;
       }
66
       @Override
       public void reset() {}

       @Override
71     public void aboutToSelfTerminate() {
       }

       public String getFileName() {
          return fileUploadName;
76     }

    }
```

Listing A.2: Sensor Capture Plugin's implementation of the framework requirements

```
public class SensorCaptureApp extends MCSStageApp {
    public static final String TYPE_PARAM = "sensor_type(s)";
    public static final String RATE_PARAM = "update_rate(s)";
    public static final String RECORD_INTERVAL = "record_interval(msec)";
    public static final String DURATION_PARAM = "duration(sec)";

    boolean finished;

    public ArrayList<Uri> output;

    public ArrayList<Integer> sensor_types;
    public ArrayList<Integer> sensor_rates;
    long record_interval;
    int duration;
    int elapsed;

    @Override
    public String getProgress() {
        return String.format("%d/%d", elapsed,duration);
    }

    @Override
    public float getProgressFloat() {
        return ((float) elapsed)/duration;
    }

    @Override
    public boolean finished() {
        return finished;
    }

    @Override
    public boolean successful() {
        return finished && elapsed >= duration;
    }

    @Override
    public void initStageSpecific(Stage s, HashMap<String, Object> params, String dyn_input) {
        duration = Integer.valueOf("" + params.get(DURATION_PARAM));
        record_interval = Integer.valueOf("" + params.get(RECORD_INTERVAL));

        String[] types = ((String)params.get(TYPE_PARAM)).split("\\s*,\\s*");
        String[] rates = ((String)params.get(RATE_PARAM)).split("\\s*,\\s*");
        if(types.length != rates.length || types.length == 0){
            throw new RuntimeException("Provided paramters were not valid");
        }

        elapsed = 0;
        output = new ArrayList<Uri>();

        sensor_types = new ArrayList<Integer>(rates.length);
        sensor_rates = new ArrayList<Integer>(rates.length);
        finished = false;
        for (int i = 0; i < rates.length; i++) {
            sensor_types.add(Integer.valueOf(types[i]));
            sensor_rates.add(Integer.valueOf(rates[i]));
        }
    }

    @Override
    public ArrayList<Uri> getOutput() {
        return output;
    }
```

```
        @Override
        public void reset() {
67          elapsed = 0;
            output = new ArrayList<Uri>();
            finished = false;
        }

72      @Override
        public void aboutToSelfTerminate() {
        }
    }
```

Listing A.3: Text Classification Plugin's implementation of the framework requirements

```
public class ClassifyApp extends MCSStageApp{

    public static final String COUNT_PARAM = "count";
    public static final String LABELS_PARAM = "labels";

    int progress;
    int count;
    String[] labels;
    boolean finished;

    String text;

    ArrayList<Uri> output = new ArrayList<Uri>(1);//will need only 1


    @Override
    public String getProgress() {
       return String.format("%d/%d", progress, count);
    }

    @Override
    public float getProgressFloat() {
       return ((float) progress)/count;
    }

    @Override
    public boolean finished() {
       return finished || progress == count;
    }

    @Override
    public boolean successful() {
       return progress >= count;
    }

    @Override
    public void initStageSpecific(Stage s, HashMap<String, Object> params, String dyn_input) {
       count = Integer.valueOf("" + params.get(COUNT_PARAM));
       labels = ((String)params.get(LABELS_PARAM)).split("\\s*,\\s*");
       //labels, unlike dynamic input, are static input, they are always the same, hence throw. Th
       if(labels.length == 0){
          throw new RuntimeException("Provided parameters were not valid");
       }

       progress = ((int) (s.getProgress() * count)); //resume progress
       output = new ArrayList<Uri>();

       text = dyn_input;
    }

    @Override
    public ArrayList<Uri> getOutput() {
       return output;
    }

    @Override
    public void reset() {
    }

    @Override
    public void aboutToSelfTerminate() {
    }

}
```

Listing A.4: Camera Capture Plugin's implementation of the framework requirements

```
1  import com.dp.mcs.shared.bean.Stage;

   public class CameraCaptureApp extends MCSStageApp {
      public static final int PICTURE_MODE = 1;
      public static final int VIDEO_MODE = 2;
6
      public static final int NO_LIMIT = -1;

      public int mode;

11    public int pic_count;
      public int video_duration;

      public int count; //TODO fix for video
      public int required_count;
16    public boolean finished;
      public String description;
      public String extention;

      public ArrayList<Uri> output;
21

      Intent getStageIntent() {
         Intent i = new Intent(getCameraIntentString());
         return i;
26    }

      public boolean finished() {
         return finished;
      }
31
      public boolean successful() {
         return finished() && count >= required_count;
      }

36    public ArrayList<Uri> getOutput() {
         return output;
      }

      public void reset(){
41       finished = false;
         count = 0;
         output = new ArrayList<Uri>();
      }


46
      public String getProgress() {
         if(required_count == NO_LIMIT)
            return Integer.toString(count);
         else
51          return String.format("%d/%d", count,required_count);
      }

      public float getProgressFloat() {
         if(required_count == NO_LIMIT)
56          return Math.abs(count);
         else
            return ((float)count)/required_count;
      }

61    private String getCameraIntentString() {
         if(mode == PICTURE_MODE)
            return MediaStore.ACTION_IMAGE_CAPTURE;
         else
```

```
            return MediaStore.ACTION_VIDEO_CAPTURE;
66      }

        public void initStageSpecific(Stage s, HashMap<String,Object> params, String dyn_input){

            int c = Integer.valueOf((String) params.get("count"));
71          String type;
            this.mode = Integer.valueOf((String) params.get("mode"));
            if(mode == PICTURE_MODE){
               pic_count = c;
               video_duration = -1;
76             type = "picture(s)";
            }
            else if (mode == VIDEO_MODE){
               video_duration = c;
               pic_count = -1;
81             type = "video(s)";
            }
            else
               throw new RuntimeException("Unexpected camera mode");

86          this.required_count = c;
            output = new ArrayList<Uri>();
            count = 0;
            description = String.format("Create %d %s", required_count, type);
            extention = (mode == PICTURE_MODE)? ".jpg":".mp4";
```

Bibliography

[1] Strategy Analytics. Strategy analytics blog. URL http://blogs.strategyanalytics.com/WSS/post/2013/01/25/Global-Smartphone-Shipments-Reach-a-Record-700-Million-Units-in-2012.aspx. Accessed: June 2013.

[2] Yoram Bachrach, Tom Minka, and John Guiver. How to grade a test without knowing the answers a bayesian graphical model for adaptive crowdsourcing and aptitude testing. *International Conference on Machine Learning*, 2012. URL http://arxiv.org/abs/1206.6386.

[3] Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, and et al. Vizwiz : Nearly real-time answers to visual questions. *Science*, 16(3):333–342, 2010. URL http://portal.acm.org/citation.cfm?id=1866080.

[4] Canalys. Canalys press release. URL http://www.canalys.com/newsroom/developing-markets-will-drive-smart-phone-market-growth-2013. Accessed: June 2013.

[5] Tathagata Das, Prashanth Mohan, Venkata N Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. *PRISM : Platform for Remote Sensing using Smartphones*, pages 63–76. ACM, 2010. URL http://research.microsoft.com/pubs/131575/mobi096-das.pdf.

[6] FasterXML. Jackson json. URL http://wiki.fasterxml.com/JacksonDocumentation. Accessed: August 2013.

[7] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[8] Raghu Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges, 2011. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6069707.

[9] Gartner. Gartner press release. URL https://www.gartner.com/newsroom/id/2335616. Accessed: June 2013.

[10] Java Serialization Group. Jvm serializers. URL https://github.com/eishay/jvm-serializers/wiki. Accessed: August 2013.

[11] P. Haghighi, S. Krishnaswamy, A. Zaslavsky, Mohamed Gaber, A. Sinha, and B. Gillick. Open mobile miner: a toolkit for building situation-aware data mining applications. *Journal of Organizational Computing and Electronic Commerce*, 2012. URL http://eprints.port.ac.uk/9054/. WNU.

[12] International Data Corporation (IDC). Idc press report. URL http://www.idc.com/getdoc.jsp?containerId=prUS23916413. Accessed: June 2013.

[13] Google Inc. Optimizing downloads for efficient network access, . URL https://developer.android.com/training/efficient-downloads/efficient-network-access.html. Accessed: August 2013.

[14] Google Inc. Android developers api guide, . URL https://developer.android.com/guide/components/index.html. Accessed: August 2013.

[15] Salil S Kanhere. Participatory sensing: Crowdsourcing data from mobile smartphones in urban spaces, 2011. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6068482.

[16] N D Lane, E Miluzzo, Hong Lu Hong Lu, D Peebles, T Choudhury, and A T Campbell. A survey of mobile phone sensing, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5560598.

[17] Walter S. Lasecki. Real-time conversational crowd assistants. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 2725–2730, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1952-2. doi: 10.1145/2468356.2479500. URL http://doi.acm.org/10.1145/2468356.2479500.

[18] Walter S Lasecki, Kyle I Murray, Samuel White, Robert C Miller, and Jeffrey P Bigham. Real-time crowd control of existing interfaces. *Proceedings of the 24th annual ACM symposium on User interface software and technology UIST 11*, page 23, 2011. URL http://dl.acm.org/citation.cfm?doid=2047196.2047200.

[19] Greg Little, Lydia B Chilton, Max Goldman, and Robert C Miller. Turkit : Human computation algorithms on mechanical turk. *New York*, 171(3):57–66, 2010. URL http://portal.acm.org/citation.cfm?id=1866040.

[20] Amazon Ltd. Amazon mechanical turk. URL https://aws.amazon.com/mturk/. Accessed: June 2013.

[21] Hong Lu, Wei Pan, Nicholas D Lane, Tanzeem Choudhury, and Andrew T Campbell. Soundsense : Scalable sound sensing for people-centric applications on mobile phones. *Architecture*, pages 165–178, 2009. URL http://portal.acm.org/citation.cfm?id=1555834.

[22] R. Meier. *Professional Android 4 Application Development*. ITPro collection. Wiley, 2012. ISBN 9781118223857. URL http://books.google.co.uk/books?id=48bnSNs_h9sC.

[23] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 323–336, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: 10.1145/1460412.1460444. URL http://doi.acm.org/10.1145/1460412.1460444.

[24] Wolfgang Pree. Meta patterns—a means for capturing the essentials of reusable object–oriented design. In *In: Proceedings of ECOOP'94*. Springer-Verlag, 1994.

[25] Moo-Ryong Ra, Bin Liu, Tom F. La Porta, and Ramesh Govindan. Medusa: a programming framework for crowd-sensing applications. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 337–350, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307668. URL http://doi.acm.org/10.1145/2307636.2307668.

[26] Vikas C Raykar. Supervised learning from multiple experts : Whom to trust when everyone lies a bit. *New York*, pages 1–8, 2009. URL http://portal.acm.org/citation.cfm?doid=1553374.1553488.

[27] Vikas C Raykar, Shipeng Yu, Linda H Zhao, Charles Florin, Luca Bogoni, and Linda Moy. Learning from crowds. *Journal of Machine Learning Research*, 11(3):1297–1322, 2010. URL http://dl.acm.org/citation.cfm?id=1859894.

[28] G. Russello, A.B. Jimenez, H. Naderi, and W. van der Mark. Poster: Firedroid: Hardening security in android with system call interposition. IEEE Symposium on Security & Privacy, 2013.

[29] P.J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012. ISBN 9780133036121. URL http://books.google.co.uk/books?id=AyY1a6-k3PIC.

[30] W. Sherchan, P.P. Jayaraman, S. Krishnaswamy, A. Zaslavsky, S. Loke, and A. Sinha. Using on-the-move mining for mobile crowdsensing. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 115–124, 2012. doi: 10.1109/MDM.2012.58.

[31] L. von Ahn. Games with a purpose. *Computer*, 39(6):92–94, 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.196.

[32] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 319–326, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985733. URL http://doi.acm.org/10.1145/985692.985733.

[33] P Welinder and P Perona. Online crowdsourcing: Rating annotators and obtaining cost-effective labels, 2010. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5543189.

[34] Jacob Whitehill, Paul Ruvolo, Tingfan Wu, Jacob Bergsma, and Javier Movellan. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. *Advances in Neural Information Processing Systems*, 22(1):1–9, 2009. URL http://mplab.ucsd.edu/ jake/OptimalLabeling.pdf.

[35] Jesse Wilson. Androids http clients. URL http://android-developers.blogspot.co.uk/2011/09/androids-http-clients.html. Accessed: August 2013.

[36] Yu Xiao, Pieter Simoens, Padmanabhan Pillai, Kiryong Ha, and Mahadev Satyanarayanan. Lowering the barriers to large-scale mobile crowdsensing. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 9:1–9:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1421-3. doi: 10.1145/2444776.2444789. URL http://doi.acm.org/10.1145/2444776.2444789.