

Medusa: A Programming Framework for Crowd-Sensing Applications *

Moo-Ryong Ra*

Bin Liu*

Tom La Porta†

Ramesh Govindan*

*Department of Computer Science
University of Southern California
{mra, binliu, ramesh}@usc.edu

†Department of Computer Science and Engineering
Pennsylvania State University
tlp@cse.psu.edu

ABSTRACT

The ubiquity of smartphones and their on-board sensing capabilities motivates *crowd-sensing*, a capability that harnesses the power of crowds to collect sensor data from a large number of mobile phone users. Unlike previous work on wireless sensing, crowd-sensing poses several novel requirements: support for humans-in-the-loop to trigger sensing actions or review results, the need for incentives, as well as privacy and security. Beyond existing crowd-sourcing systems, crowd-sensing exploits sensing and processing capabilities of mobile devices. In this paper, we design and implement Medusa, a novel programming framework for crowd-sensing that satisfies these requirements. Medusa provides high-level abstractions for specifying the steps required to complete a crowd-sensing task, and employs a distributed runtime system that coordinates the execution of these tasks between smartphones and a cluster on the cloud. We have implemented ten crowd-sensing tasks on a prototype of Medusa. We find that Medusa task descriptions are two orders of magnitude smaller than standalone systems required to implement those crowd-sensing tasks, and the runtime has low overhead and is robust to dynamics and resource attacks.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: [Language Constructs and Features – Frameworks]

General Terms

Design, Experimentation, Human Factors, Reliability, Security

Keywords

Crowd-sensing, Macroprogramming, Programming Framework, Smartphone, Worker Mediation

*The first author, Moo-Ryong Ra, was supported by Annenberg Graduate Fellowship. This research was partially sponsored by the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053 and by the USC METRANS Transportation Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.
Copyright 2012 ACM 978-1-4503-1301-8/12/06 ...\$10.00.

1. INTRODUCTION

The ubiquity of smartphones and other mobile devices, and the plethora of sensors available on them, have inspired innovative research that will, over time, lead to sophisticated context-aware applications and systems. Some of this research has explored ways in which mobile phone users can *contribute* sensor data towards enabling self-reflection, environmental awareness, or other social causes.

In this paper, we consider a specific form of acquiring sensor data from multiple smartphones that we call *crowd-sensing*. Crowd-sensing is a capability by which a *requestor* can recruit smartphone users (or *workers*) to provide sensor data to be used towards a specific goal or as part of a social or technical experiment (e.g., *tasks* such as forensic analysis, documenting public spaces, or collaboratively constructing statistical models). Workers' smartphones collect sensor data, and may process the data in arbitrary ways before sending the data to the requestor.

Crowd-sensing is a form of networked wireless sensing, but is different from prior research on networked sensing for two reasons: each smartphone is owned by an individual, and some sensing actions on the smartphone may require human intervention (e.g., pointing a camera at a specific target, or initiating audio capture at the appropriate moment). Crowd-sensing is different from crowd-sourcing in one important respect: a crowd-sourcing system like the Amazon Mechanical Turk (AMT [1]) permits requestors to task participants with human intelligence tasks like recognition or translation, while a crowd-sensing system enables requestors to task participants with acquiring processed sensor data.

For these reasons, any software system designed to support crowd-sensing is qualitatively different from systems for wireless sensing or crowd-sourcing. Crowd-sensing systems must support ways in which workers can be incentivized to contribute sensor data, since participating in crowd-sensing may incur real monetary cost (e.g., bandwidth usage). In some cases, it may be necessary to support “reverse” incentives, where workers pay for the privilege of participating in the requestor's task. They must also support *worker-mediation*, by which (human) workers can mediate in the sensor data collection workflows. Specifically, humans may trigger sensing actions, and annotate collected sensor data. In addition, a crowd-sensing system must support privacy of sensor data contributions, and worker anonymity (in cases where workers desire anonymity). Identifying the requirements for crowd-sensing is our first important contribution (Section 2).

In this paper, we tackle the challenge of *programming* crowd-sensing tasks. A high-level programming language can simplify the burden of initiating and managing crowd-sensing tasks, especially for non-technical requestors who we assume will constitute the majority of crowd-sensing users. Our second contribution is

the design of a high-level programming language called *MedScript* and an associated runtime system called *Medusa* for crowd-sensing (Section 4). *MedScript* provides abstractions called *stages* for intermediate steps in a crowd-sensing task, and for control flow between stages (called *connectors*). Unlike most programming languages, it provides the programmer language-level constructs for incorporating workers into the sensing workflow. The *Medusa* runtime is architected (Section 3) as a partitioned system with a cloud component and a smartphone component. For robustness, it minimizes the task execution state maintained on the smartphone, and for privacy, it ensures that any data that leaves the phone must be approved by the (human) worker. Finally, it uses AMT to recruit workers and manage monetary incentives.

Our third contribution is an evaluation of *Medusa* on a complete prototype (Section 5). We use our prototype to illustrate that *Medusa* satisfies many of the requirements discussed above. We demonstrate that *Medusa* can be used to compactly express several novel crowd-sensing tasks, as well as sensing tasks previously proposed in the literature. For three different tasks, *Medusa*'s task descriptions are almost two orders of magnitude more compact than stand-alone implementations of those tasks. *Medusa*'s overhead is small, and it provides effective sandboxing against unauthorized access to smartphone resources as well as to excessive resource usage.

Medusa represents a convergence of two strands of research (Section 6): participatory sensing systems like [16] and crowd-sourcing frameworks like [1]. Unlike the former, *Medusa* explicitly supports incentives and worker-mediation; unlike the latter, *Medusa* enables requestors to gather raw or processed sensor data from mobile devices in a substantially automated way.

2. CROWD-SENSING: MOTIVATION AND CHALLENGES

Motivation. Consider the following scenario. A social science researcher, Alice, is interested in studying the dynamics of “Occupy” movements across the globe. Her methodology is to obtain ground truth videos about life in “Occupy” encampments across the United States and overseas. This methodology of obtaining observations of human life in urban spaces was pioneered by William Whyte [20] and is accepted practice in public planning.

However, it is logistically difficult for Alice to visit each of these locations to conduct research, so she resorts to an idea that we call *crowd-sensing*, which leverages the prevalence of mobile devices with sensors and the power of crowds. She recruits volunteers at each one of the encampments by offering them small monetary incentives to provide her with videos of daily life at the camps. Some volunteers take one or more videos on a mobile device, and upload summaries of each video (e.g. a few frames or a short clip) to an Internet-connected service. Other volunteers upload summaries of videos they may have taken before being recruited. Alice obtains a large corpus of summaries, which she makes available to other volunteers to *curate*, selecting only the relevant subset that may be useful for research. She then asks the volunteers whose summaries have been chosen to upload the full videos, and pays them for their participation. She analyzes the curated set of videos, and draws conclusions about similarities and differences between “Occupy” communities across the globe.

In the scenario above, we say that Alice is interested in conducting a *task*, that of collecting relevant videos for her research. A task is initiated by a *requestor*. Each task consists of several *stages*. In our example, the stages were, in order: recruiting volunteers, volunteers taking videos on their smartphones, uploading summaries,

curating summaries using another set of volunteers, and uploading the full videos. Volunteers who participate in tasks are called *workers*; workers have smartphones¹ that are used to perform one or more stage actions.

In this paper, we consider the architectural requirements, the design of a high-level programming framework and an associated runtime system for crowd-sensing. In the absence of such a programming framework, Alice may need to manually recruit volunteers (e.g., by advertising in mass media or using her social network), follow-up with the volunteers to upload summaries and videos, manually recruit volunteers for curation (e.g., students in her class), implement software that simplifies the curation process, and ensure that payments are made to all volunteers.

Ideally, our programming framework should abstract all these details, and allow Alice in our scenario to compactly express the stages in her task. The associated runtime should automate stage execution, relieving the requestor from the burden of manually managing stages.

There are many other examples of crowd-sensing, beyond video documentation:

Spot Reporter A journalist is under deadline pressure to write an article about a developing forest fire and with its impact on nearby communities. He recruits volunteers with smartphones to send pictures of the extent of fire damage, interview local residents, and report on the efficacy of the first responders.

Auditioning A television station invites budding actors to submit videos of their acting skills, in order to select actors for a second stage of in-person auditioning for an upcoming television show.

Collaborative Learning A software company is developing a lifestyle monitoring mobile app, which lets users self-reflect on their activities. It would like to recruit volunteers to submit labeled samples of accelerometer readings; these are used to build a machine-learning classifier of human activity which will be embedded in the mobile app.

Forensic Analysis Security officials at a stadium are investigating an outbreak of violence and would like to determine people who were at the scene of the outbreak when the violence occurred. They request volunteers, who may have taken pictures during the event, to send snapshots of faces in the stadium.

We describe these tasks in detail later in Section 5.

Requirements. A practical programming framework that supports crowd-sensing tasks must satisfy several challenging requirements. Some of these requirements govern the expressivity of the programming language, while others constrain the underlying runtime system. We discuss these in order.

Expressivity Requirements. Requestors must be able to specify *worker-mediation*, i.e., a worker may need to perform an action to complete a stage such as initiating or stopping the recording of the video or audio clip, labeling data or approving data for upload (see below). However, not all stages will require workers to initiate sensor data collection; requestors may want to design stages that *access stored sensor data*, such as in the forensic analysis task.

Stages must support not just sensing, but also *in-network processing*; this helps conserve network bandwidth for users with data usage limits or reduces upload latency by reducing the volume of

¹Henceforth, we will use the term smartphones, although our designs apply equally well to other mobile devices such as tablets.

information that needs to be transmitted on a constrained connection. For example, in our video documentation example, video summaries were first extracted from the video in order to weed out irrelevant videos. Similarly, feature vectors may be extracted in the collaborative learning task, or faces in the forensic analysis task. In addition, the programming language must be *extensible*, in order to support new sensors or new in-network processing algorithms.

Tasks may have *timeliness* requirements and any contribution received after the deadline are discarded; in general, we anticipate task deadlines to be on the order of hours or days after task initiation, so traditional quality-of-service concerns may not apply. Tasks may require *crowd-curation*; in our scenario, Alice relies on this to pre-select videos likely to be most relevant to her research objectives.

Requestors must be able to specify monetary *incentives* for workers. In our scenario, Alice promises to pay participants for uploading videos. Some tasks may also require *reverse-incentives*, where the workers pay the requestors for the privilege of participating in the task. In our auditioning example, a requestor may charge a small fee in order to disincentivize frivolous participants.

Runtime Requirements. Workers must be able to sign up for *multiple concurrent tasks*, and requestors should be able to initiate multiple tasks concurrently. Thus, a smartphone user may wish to participate in collaborative learning, but may also sign up to be a spot reporter. Stages in a task may be executed at different times by different workers; thus, stage execution is *not necessarily synchronized* across workers. In the video documentation task, one worker may upload their summaries several hours after another worker has uploaded his or her video. Task execution must be *robust* to intermittent disconnections or longer-term outages on smartphones.

The runtime should preserve subject *anonymity* with respect to requestors, and should contain mechanisms for ensuring data *privacy*. It should use best practices in *secure communication*, and ensure the *safety* of execution on worker smartphones. We discuss later what kinds of anonymity, privacy, security, and safety are achievable in our setting. Finally, the runtime must respect user-specified *resource usage policies* on the smartphone for crowd-sensing; for example, if generating a video summary is likely to consume significant battery power, the framework must terminate the stage or take other actions discussed later.

Summary. This is a fairly complex web of requirements, but we believe they are necessary for crowd-sensing. Worker-mediation and crowd-curation leverage human intelligence to ensure high-quality sensor data. Support for concurrent tasks and asynchrony enables high throughput of tasks. Monetary incentives, as well as anonymity, privacy and security features as well as resource controls incentivize worker participation. Ensuring task execution robustness in the runtime relieves programmers of the burden of dealing with failures.

In the next section, we describe an architecture that meets these requirements.

3. MEDUSA SYSTEM ARCHITECTURE

In this section, we first describe the architectural principles that guide the design of our crowd-sensing programming system, called Medusa. We then illustrate our design (Figure 1) by describing how the video documentation crowd-sensing task is executed by Medusa.

3.1 Architectural Principles

Medusa is a high-level language for crowd-sensing. In Medusa, programmers specify crowd-sensing tasks as a sequence of stages.

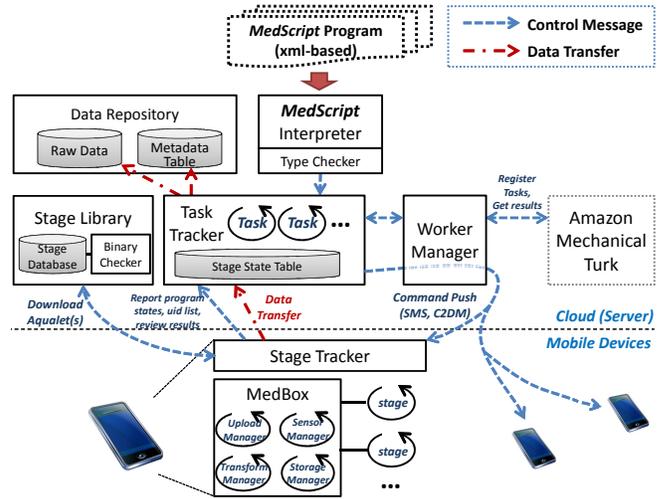


Figure 1: System Architecture

Thus, Alice would program her video documentation task using a description that looks approximately like this:

```
Recruit -> TakeVideo -> ExtractSummary
-> UploadSummary -> Curate -> UploadVideo
```

This describes the sequence of steps in the task. The Medusa runtime executes these tasks. The design of the Medusa runtime is guided by three architectural decisions that simplify overall system design.

Principle #1: Partitioned Services. Medusa should be implemented as a partitioned system that uses a collection of services both on the cloud and on worker smartphones. This constraint follows from the following observation. Some of the requirements are more easily and robustly accomplished on an always Internet-connected cloud server or cluster: task initiation, volunteer recruitment, result storage, and monetary transactions. Others, such as sensing and in-network processing, are better suited for execution on the smartphone.

Principle #2: Dumb Smartphones. Medusa should minimize the amount of task execution state that is maintained on smartphones. This design principle precludes, for example, large segments of a task from being completely executed on the smartphone without the cloud being aware of execution progress. We impose this constraint to enable robust task execution in the presence of intermittent connectivity failures, as well as long-term phone outages caused, for example, by workers turning off their smartphones.

Principle #3: Opt-in Data Transfers. Medusa should automatically require a user's permission before transferring any data from a smartphone to the cloud. Data privacy is a significant concern in crowd-sensing applications. Our principle ensures that, at the very least, users have the option to opt-out of data contributions. Before workers opt-in, they may view a requestor's privacy policy.

Discussion. Other designs of the runtime are possible. For example, it might be possible to design a purely peer-to-peer crowd-sensing system, and it might also be possible to empower smartphones to exclusively execute crowd-sensing tasks. In our judgement, our architectural principles enable us to achieve all of the requirements outlined in the previous section without significantly complicating system design. Furthermore, these principles do not precisely determine exactly what functionality to a place on the

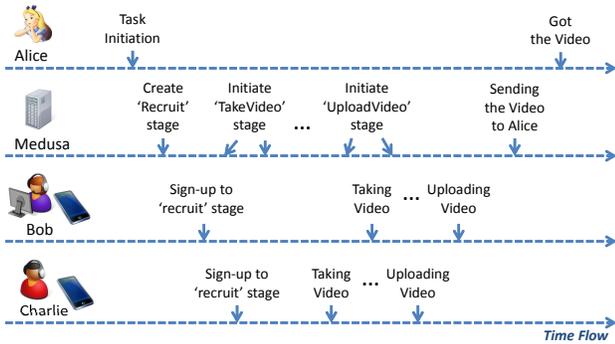


Figure 2: Illustration of video documentation task

cloud or on the phone: our design is one instantiation that adheres to these principles.

Some of these principles may appear to conflict each other. For example, if the smartphone components are designed to be “dumb”, then intermediate data products during task execution must be communicated to the cloud, which may require significant user opt-in interactions and hamper usability. We describe later how we address this tension.

Finally, our opt-in requirement can impact usability by requiring user intervention whenever data leaves the phone. In the future, we may consider relaxing this constraint in two ways. First, users may be willing to sacrifice privacy for convenience or monetary reward. In the future, we may consider policies that allow users to opt-out of approving data transfers for tasks which promise significant monetary incentives, for example. An understanding of these kinds of exceptions can only come from experience using the system. Second, some of the in-network processing algorithms may be designed to be privacy-preserving (e.g., by using differential privacy [8]), so user assent may not be necessary for data generated by these algorithms.

3.2 How Medusa Works

In keeping with our *partitioned services* principle, Medusa is structured as a collection of services running on the cloud and on the phone (Figure 1). These services coordinate to perform crowd-sensing tasks. We give a high-level overview of Medusa by taking our running example of Alice’s video documentation task.

In Medusa, Alice writes her video documentation task in MedScript, a high-level language designed for non-experts, which provides stages as abstractions and allows programmers to express control flow between stages. She submits the program to the MedScript interpreter, which is implemented as a cloud service. Figure 2 illustrates the sequence of actions that follows the submission of the task to Medusa.

The interpreter parses the program and creates an intermediate representation which is passed on to a Task Tracker. This latter component is the core of Medusa, coordinating task execution with other components as well as with workers’ smartphones. For the `Recruit` stage, Task Tracker contacts Worker Manager (a back-end service), which initiates the recruitment of workers. As we shall describe later, Worker Manager uses Amazon Mechanical Turk [1] for recruitment and a few other tasks. When workers agree to perform the task, these notifications eventually reach Task Tracker through Worker Manager; different workers may agree to perform the task at different times (e.g., Bob and Charlie in Figure 2). More generally, workers may perform different stages at

different times. Monetary incentives are specified at the time of recruitment.

Once a worker has been recruited, the Task Tracker initiates the next stage, `TakeVideo`, on that worker’s smartphone by sending a message to the Stage Tracker, one instance of which runs on every phone. The `TakeVideo` stage is a downloadable piece of code from an extensible Stage Library, a library of pre-designed stages that requestors can use to design crowd-sending tasks. Each such stage executes on the phone in a sandboxed environment called MedBox. The `TakeVideo` stage requires human intervention – the worker needs to open the camera application and take a video. To remind the worker of this pending action, the stage implementation uses the underlying system’s notification mechanism to alert the worker. Once the video has been recorded, Stage Tracker notifies the Task Tracker of the completion of that stage, and awaits instructions regarding the next stage. This is in keeping with our *dumb smartphones* principle. The video itself is stored on the phone.

The Task Tracker then notifies the Stage Tracker that it should run the `ExtractSummary` stage. This stage extracts a small summary video comprised of a few sample frames from the original video, then uploads it to the Task Tracker. Before the upload, the user is required to preview and approve the contribution, in keeping with our *opt-in data transfers* principle.

Execution of subsequent stages follows the same pattern: the Task Tracker initiates the `curate` stage while a completely different set of volunteers rates the videos. Finally, the selected videos are uploaded to the cloud. Only when all stages have completed is Alice notified (Figure 2).

Medusa maintains persistent state of data contributions in its Data Repository as well as the state of stage execution for each worker in its Stage State Table. As we shall discuss later, this enables robust task execution.

In the next section, we discuss each of these components in greater detail.

4. MEDUSA DESIGN

Medusa achieves its design objectives using a cloud runtime system which consists of several components, together with a runtime system on the smartphone. Before we describe these two subsystems, we begin with a description of the MedScript programming language.

4.1 The MedScript Programming Language

The MedScript programming language is used to describe crowd-sensing tasks, so each task described in Section 2 would be associated with a corresponding MedScript. More precisely, a MedScript defines a *task instance*, specifying the sequence of actions performed by a single worker towards that crowd-sensing task. In general (as described in the next section), requestors may ask the Medusa runtime to run multiple instances of a MedScript task at different workers, and a single worker may also run multiple task instances. Moreover, a single worker may concurrently run instances of different tasks (possibly) initiated by different requestors.

Listing 1 shows the complete listing of the video documentation task. We use this program listing to illustrate various features of the MedScript language.

MedScript consists of two high-level abstractions: *stages* and *connectors*. A stage (e.g., line 8) describes either a sensing or computation action, or one that requires human-mediation, and connectors (e.g., line 65) express control flow between stages. In our current realization, MedScript is an XML-based domain-specific language; we have left it to future work to explore a visual programming front-end to MedScript.

```

1 <xml>
2 <app>
3   <name>Video-Documentation </name>
4   <rrid>[User's Requestor ID]</rrid>
5   <rrkey>[User's Requestor Key]</rrkey>
6   <deadline>21:00:00 12/16/2011</deadline>
7
8   <stage>
9     <name>Recruit </name> <type>HIT</type>
10    <binary>recruit </binary>
11    <config>
12      <stmt>Video Documentation App. Demonstration</stmt>
13      <expiration>21:00:00 12/16/2011</expiration>
14      <reward>.05</reward>
15      <output>W_WID</output>
16    </config>
17  </stage>
18  <stage>
19    <name>TakeVideo </name> <type>SPC</type>
20    <binary>mediagen </binary>
21    <trigger>user-initiated </trigger> <review>none </review>
22    <config>
23      <params>-t video </params>
24      <output>VIDEO</output>
25    </config>
26  </stage>
27  <stage>
28    <name>ExtractSummary </name> <type>SPC</type>
29    <binary>videosummary </binary>
30    <trigger>immediate </trigger> <review>none </review>
31    <config>
32      <input>VIDEO</input>
33      <output>SUMMARY</output>
34    </config>
35  </stage>
36  <stage>
37    <name>UploadSummary </name> <type>SPC</type>
38    <binary>uploaddata </binary>
39    <trigger>immediate </trigger>
40    <config>
41      <input>SUMMARY</input>
42    </config>
43  </stage>
44  <stage>
45    <name>Curate </name> <type>HIT</type>
46    <binary>vote </binary> <wid>all </wid>
47    <config>
48      <stmt>Judging from Video Summaries</stmt>
49      <expiration>21:00:00 12/16/2011</expiration>
50      <reward>.01</reward>
51      <numusers>2</numusers>
52      <input>SUMMARY</input>
53      <output>SSMASK</output>
54    </config>
55  </stage>
56  <stage>
57    <name>UploadVideo </name> <type>SPC</type>
58    <binary>uploaddata </binary>
59    <trigger>immediate </trigger>
60    <config>
61      <input>SSMASK, VIDEO</input>
62    </config>
63  </stage>
64
65  <connector>
66    <src>Recruit </src>
67    <dst> <success>TakeVideo </success> <failure>Recruit </failure> </dst>
68  </connector>
69  <connector>
70    <src>TakeVideo </src>
71    <dst> <success>ExtractSummary </success> <failure>Recruit </failure> </dst>
72  </connector>
73  <connector>
74    <src>ExtractSummary </src>
75    <dst> <success>UploadSummary </success> <failure>Recruit </failure> </dst>
76  </connector>
77  <connector>
78    <src>UploadSummary </src>
79    <dst> <success>Curate </success> <failure>Recruit </failure> </dst>
80  </connector>
81  <connector>
82    <src>Curate </src>
83    <dst> <success>UploadVideo </success> <failure>Recruit </failure> </dst>
84  </connector>
85 </app>
86 </xml>

```

Listing 1: Video Documentation

Stages. A stage is a labeled elemental instruction in MedScript. Thus, `TakeVideo` is a label for the stage described starting on line 18. Stage labels are used in connector descriptions (see below).

Medusa defines two types of stages: a *sensing-processing-communication* stage or (SPC stage), and a *human-intelligence task* stage (or HIT stage). An SPC stage extracts sensor data from either a physical sensor (such as a camera, GPS, or accelerometer) or a logical sensor (such as a system log or a trace of network measurements), processes sensor data to perform some kind of summarization or recognition, or communicates sensor data to the re-

questor. A HIT stage exclusively requires human review and/or input and does not involve sensing or autonomous data processing. Thus, for example, `TakeVideo` or `ExtractSummary` or `UploadVideo` (lines 18, 27, and 56) are all examples of SPC stages, while `Recruit` and `Curate` (lines 8 and 44) are examples of HIT stages. Stages may take parameters; for example, the reward parameter for the `Recruit` stage specifies a monetary reward (line 14).

Each stage has an associated stage binary, referenced by the binary XML tag (e.g., line 10). Medusa maintains an extensible library of stage binaries in its Stage Library. Stage binaries may be re-used across tasks, and also within the same task: both the `UploadSummary` and `UploadVideo` stages use the `uploaddata` binary.

Connectors. Each stage can have one of two outcomes: *success* or *failure*. A stage may fail for many reasons: if a stage does not encounter failure, it is deemed to be successful. For each stage, *connectors* specify *control-flow* from that stage to the next stage upon either success or failure. Thus, in our example, if `TakeVideo` is successful (lines 69-72), control flow transfers to `ExtractSummary`, else control flow reverts to the `Recruit` stage.

In general, for each outcome (success or failure), there is at most one target next stage. If a target is not specified for an outcome, the Medusa runtime terminates the corresponding task instance.

Medusa allows a special control-flow construct called the *fork-join*. From one of the outcomes of the stage (usually the success outcome), a programmer may define multiple connectors to different target stages. This indicates that the target stages are executed *concurrently*. This construct is useful, for example, to concurrently read several sensors in order to correlate their readings. We give a detailed example of this construct later. Medusa enforces, through compile-time checks, that *both* the success and failure outcomes of these target stages are connected to (or *join*) a single stage, one which usually combines the results of the target stages in some way.

Data Model and Stage Inputs and Outputs. Each stage in Medusa produces exactly one output. This output is explicitly assigned by the programmer to a *named variable*. In our example, named variables include `VIDEO` and `SUMMARY`. The scope of a named variable is the task instance, so that any stage in an instance may access the named variable. If two instances execute on the same device, their variable namespaces are distinct.

Each stage can take zero or more inputs. Inputs to stages are defined by variables. The `ExtractSummary` stage takes `VIDEO` as input (line 32). Each named variable has an associated type (see below), and the MedScript compiler performs static type checking based on the known types of stage outputs.

In general, a named variable's value is a bag of one or more *homogeneously-typed* items, and the named variable is said to have the corresponding type. Supported types in Medusa include integer, floating-point, and many predefined *sensor data types*. Each sensor data type represents a contiguous sequence of sensor data, and examples include: a video clip, an audio clip, a contiguous sequence of accelerometer readings, and so on.

Medusa does not define an assignment operator, so variables can only be instantiated from stage outputs. For example, `VIDEO` is instantiated as the output of `TakeVideo` (line 24) and `SUMMARY` as the output of `ExtractSummary` (line 33). Both of these variables are bags of video clips: the former contains videos taken by the user and the latter the summary videos generated by `ExtractSummary`.

Stages and User-Mediation. One of the distinguishing aspects of Medusa is its support for worker-mediation. Workers can be in-the-loop in four ways.

First, the HIT stages `Recruit` and `Curate` require user intervention. Workers have to agree to perform a task instance during the `Recruit` stage. Requestors can specify monetary incentives as well as a deadline for recruitment. The `Curate` stage permits humans to review sensor data submissions and select submissions relevant to the task at hand. This takes as input a named variable containing one or more data items (in our example `SUMMARY`, line 52), and produces as output a bit mask (`SMASK`, line 53), that indicates selected submissions. This bitmask can be used in later stages to determine, for example, which data items to upload.

Second, only one stage binary, `uploaddata`, can transfer data from the phone to the cloud and, before data is uploaded, the worker is presented with a *preview* of the data and is given an opportunity to opt-in to the submission. Optionally, at this stage, the requestor may specify a link to a privacy policy that describes how workers' submissions will be used. If a worker chooses to opt out of the upload, the `uploaddata` stage fails.

Third, some stages, such as the `TakeVideo` stage, may require explicit worker action in order to *initiate* sensing. This form of worker-mediation leverages worker intelligence to determine the appropriate time, location, and other environmental conditions to take sensor data samples. This is indicated by a `trigger` parameter to a stage (e.g., line 21).

Fourth, requestors may provide humans with the option to *annotate* the output of any stage. Once a stage has produced an output, users are presented with a prompt to label the output by either selecting from a list of choices or by adding freeform text annotations. These annotations are stored as metadata with the data items. This capability is useful in our collaborative learning application, where users submit labeled training samples that, for example, indicate activities (like walking, running, sitting).

Task Parameters. In addition to stage parameters, programmers may specify *task parameters*, usually before the stage and connector definitions. In our example, the requestor specifies a deadline for the task. If a task instance is not completed before its deadline, it is deleted. Other parameters can include requestor credentials, described below.

Failure Semantics. As described above, each stage has two possible outcomes: success or failure. Stages can fail for several reasons. The smartphone may be turned off by its owner or its battery may die. The deadline on a stage expires because a worker failed either to initiate a sensing action or to annotate a sensor result within the deadline. The merge or join stage after a fork may fail, because one of the forked stages failed. Finally, stages may fail because components of the runtime fail.

Deadline and runtime failures result in task instances being aborted. However, other failures may result in one or more stages being re-tried; the requestor can specify, using appropriate connectors, which stages should be retried upon failure.

4.2 Medusa Cloud Runtime

The Medusa cloud runtime consists of several components described below.

MedScript Interpreter. Requestors submit their XML task descriptions to the MedScript interpreter and indicate how many instances of tasks should be run by Medusa. For example, if Alice wishes to receive about 50 videos in the hope that about 10-15 of them are of sufficient quality for her research, she can indicate to the interpreter that she wishes to spawn 50 instances. Requestors can add additional instances if their initial estimates proved to be incorrect.

The interpreter performs compile-time checks on the submit-

ted task descriptions: static type checks, checks on restrictions for fork-join connectors, and sanity checks on named variables. It then converts the task description into an intermediate representation and stores this in the Stage State Table, together with the number of instances to be run. Finally, it notifies the Task Tracker, which is responsible for coordinating the stage execution for all instances of this task.

Task Tracker. In Medusa, each submitted task is associated with a fresh instance of a Task Tracker. The Task Tracker spawns multiple worker instances and keeps track of their execution. Recall that different stages may execute at different times at different workers. The Task Tracker creates *instance state* entries that store the current state of execution of each instance in the Stage State Table, a persistent store.

For HIT stages, the Task Tracker uses the Worker Manager to initiate those stages. Thus, if Alice wishes to invoke 50 instances of her video documentation task, the Task Tracker first instantiates 50 instances of the `Recruit` stage using Worker Manager. When a worker signs up for one of the task instances, the Worker Manager notifies the Task Tracker, which then proceeds to instantiate the `TakeVideo` stage on the worker's smartphone.

To do this, the Task Tracker notifies the Stage Tracker on the worker's smartphone to begin executing the corresponding stage. We describe the notification mechanism later in this section. When that stage is completed on the smartphone, the stage tracker returns a list of *references* (pointers) to data items in that stage's output variable; the data items themselves are stored on the smartphone. These references are stored as part of the instance state; if the smartphone is turned off immediately after the `TakeVideo` stage, the Task Tracker has all the information necessary to execute the subsequent stage when the smartphone restarts.

More generally, the Task Tracker notifies the Stage Tracker instructing it to execute an SPC stage on the phone. It passes the references corresponding to input variables for that stage. At the end of the execution, the Stage Tracker sends the Task Tracker all references for data items in the output of that stage.

Thus, in Medusa, the Task Tracker coordinates the execution of every stage and maintains instance state information in persistent storage. A single worker may concurrently sign up for multiple instances of the same task, and/or instances of many different tasks. If a Task Tracker fails, it can be transparently restarted and can resume stage execution at each worker. Between the time when the Task Tracker fails, and when it resumes, some stages may fail because the Stage Tracker may be unable to contact the Task Tracker; these stages will usually be re-executed, depending upon the control flow specified by the requestor.

The Task Tracker keeps track of the task deadline; when this expires, the Task Tracker terminates the corresponding worker instance by updating the program state appropriately. The Task Tracker also keeps internal timeouts for stage execution, and may decide to retry stages whose timeout expires. This ensures robustness to smartphone outages and connectivity failures, as well as to humans who may not have initiated sensing actions on time or completed an annotation of sensor data.

The Task Tracker does not know the identities of the requestor or the worker, instead referring to them in its internal data structures using an opaque machine-generated ID. Below, we describe how these IDs are mapped to identities of individual workers or of the requestor. This design ensures that knowledge about identities of participants is localized to a very small part of the system (described later), thereby reducing the vulnerability footprint.

The Task Tracker thus supports concurrent execution of worker instances, tracks unsynchronized stage execution across different

workers, enforces timeliness of stage execution, as well as ensures robustness.

Worker Manager. The Worker Manager supports HIT stage execution, monetary transactions, crowd-curation, and worker smartphone notifications. To achieve these objectives, Worker Manager uses the Amazon Mechanical Turk (AMT [1]) system as a backend. AMT was designed to support crowd-sourcing of human intelligence tasks, such as translation, behavioral surveys, and so forth.

When the Task Tracker encounters a `Recruit` stage, it invokes the Worker Manager and passes to it the monetary incentive specified in the stage description, together with the number of desired instances. The Worker Manager posts AMT tasks (using the AMT API) seeking workers who are interested in signing up for the crowd-sensing task. Usually, requestors will include an optional URL as a parameter to the `Recruit` stage so that potential workers can see a description of the work involved. Workers may use a browser on their smartphone to sign up on AMT for this task instance. When a worker signs up, AMT notifies the Worker Manager, which in turn notifies the Task Tracker.

In order to post a task on AMT, the Worker Manager needs to be entrusted with an Amazon-issued ID and key for the requestor. These requestor credentials are specified as task parameters. However, the ID and key do not reveal the identity of the requestor. Thus, while requestor identity is not exposed, limited trust must be placed in the Worker Manager not to misuse IDs and keys.

When a task instance completes (e.g., when a worker uploads a selected video in our video documentation example), the Task Tracker contacts Worker Manager, which, in turn, presents the uploaded data to the requestor and seeks approval. Once the requestor approves the submission, money is transferred from the requestor’s AMT account to the worker’s AMT account.

Crowd-curation works in a similar manner. The Worker Manager spawns an AMT “voting” task inviting volunteers to vote on submissions in return for a monetary incentive. When the voting is complete, AMT notifies the Worker Manager, which in turn notifies the Task Tracker.

The Worker Manager also supports reverse-incentives, so workers can pay requestors for the privilege of contributing sensor data. Our auditioning example requires this, where the reverse incentive is designed to dissuade frivolous participation. The Worker Manager achieves this by posting a task requiring a null contribution on AMT with roles reversed: the worker as the “requestor” and the requestor as the only permitted “worker”. As soon as the requestor signs up and the worker “accepts” the contribution, the requisite payment is transferred from the workers AMT account to that of the requestor.

Our use of AMT requires workers and requestors to have AMT accounts and to manage their balances on these accounts. Both workers and requestors have an incentive to do this: requestors are able to complete tasks using the system that they might not otherwise have been able to, and workers receive micro-payments for completing sensing tasks that require little cognitive load (since many of the steps can be automated).

Moreover, AMT preserves *subject anonymity* so that worker identities are not revealed to requestor (unless workers choose to). In Medusa, this subject anonymity is also preserved within the rest of the system (the runtimes on the smartphone and cloud) by identifying requestors and workers only using the opaque AMT IDs. However, since Medusa supports crowd-sensing, the sensing data submitted by a worker may reveal the identity of the worker. By opting-in to an upload of the sensor data, workers may give up subject anonymity; they can always choose to opt-out, or carefully ini-

tiate sensor data collection (e.g., taking videos whose content does not reveal the sender), should they wish to retain anonymity.

Finally, because workers are anonymous to the rest of the system, and AMT is the only component that knows the identity of the worker, the Task Tracker uses AMT (through the Worker Manager) to notify workers’ smartphones of SPC stage initiation. This notification is done by using AMT to send an SMS message to the smartphone, which is intercepted by the Stage Tracker. That component initiates stage execution on the phone. When an SPC stage completes, the Stage Tracker directly transfers program states to the Task Tracker without AMT involvement. However, this transfer contains enough identifying information to uniquely identify the task instance.

Stage Name	Description
probedata	Searches for data items matching specified predicates.
uploaddata	Uploads data corresponding to specified input variable.
mediagen	Invokes external program to sense a data item (e.g., video clip or audio clip).
videosummary	Generates a motion jpeg summary from a original high-quality video data.
facetect	Extracts faces from videos or image inputs
netstats	Scans Bluetooth and Wifi interfaces. Users can set filters to get selective information.
vcollect	Collects samples directly from a sensor (e.g., from accelerometer or microphone).
vfeature	Extracts feature vectors from input raw sensor data.
gpsrawcollect	Extracts sequence of location samples at specified rate.
combiner	Temporally merges two sensor data streams.
cfeature	Extracts feature vectors from temporally merged streams.

Table 1: Stage Binaries in Stage Library.

Stage Library. Stage Library is a reusable and extensible library of SPC stage binary implementations. For example, the binaries for the `TakeVideo`, `ExtractSummary`, and `UploadVideo` stages (namely `mediagen`, `videosummary` and `uploaddata`) are stored here. These binaries are stored on the cloud, but when a stage is initiated on a smartphone, the Stage Tracker is responsible for downloading the binaries. Stage binaries may be cached on smartphones, and caches are periodically purged so that the latest versions of the stage binaries may be downloaded. Table 1 gives some examples of stages that we have implemented.

The Stage Library ensures Medusa’s extensibility. When a requestor needs a specific functionality to be implemented as a stage binary, they may provide this implementation themselves or (more likely) make a feature request to the Medusa system developers. A larger-scale deployment, which is left to future work, will provide us with insights into how often a new stage needs to be added to Stage Library. We anticipate that there will likely be many such additions to the Stage Library initially, but, with time, requestors will be able to reuse stages frequently. To support the Stage Library’s extensibility, stages are stored on the cloud and downloaded on demand to the phone.

Before stage binaries are admitted into the Stage Library, they are *vetted* for safety. As we describe below, stages are only allowed to invoke a restricted set of functionality on the smartphone. For example, stages cannot access the underlying file system on the phone or transmit data to arbitrary websites. Medusa ensures this by using static program analysis to determine the set of APIs in-

voked by stage binaries; it conservatively rejects any stage binary that accesses disallowed APIs or uses dynamic API binding.

In this way, Medusa achieves safety and extensibility of stages that implement sensing, in-network processing, and communication functions.

4.3 Medusa Runtime on the Smartphone

The Medusa runtime on the smartphone consists of the two components described below.

Stage Tracker. As described above, the Stage Tracker receives stage initiation notifications by intercepting SMS messages. Each SMS message contains an XML description of the stage; the Stage Tracker parses this description, downloads the stage binary if necessary, and initiates the execution of the stage. The SMS message also contains the values of named variables instantiated during previous stage computations; recall that these contain pointers to data items stored on the phone. The Stage Tracker is responsible for setting up the namespace (containing these named variables and their values) before the stage executes. It is also responsible for implementing the triggers necessary to initiate stage execution (e.g., to start recording a video clip), as well as the human-mediation required for annotations.

Once a stage’s execution is complete, stage state is transferred back to the Stage State Table. As discussed before, this includes only references to data items, not actual data (the one exception is the `upload` stage, which may transfer data but only after the worker explicitly opts in). Task Tracker polls the Stage State Table periodically to determine stage completion.

If a stage execution fails for any reason (e.g., resource constraints or implementation bugs in a stage), the Stage Tracker returns a failure notification to the Task Tracker. If the Task Tracker does not hear back from the Stage Tracker within a timeout period, it tries to contact the Stage Tracker before determining whether the stage has failed or not.

By design, the Stage Tracker does not know (or care) to which instance of which task a given stage belongs. Furthermore, the Stage Tracker can initiate multiple concurrent stages. The Stage Tracker itself is stateless, and transfers all intermediate program state to the Task Tracker which maintains all instance state. If a Stage Tracker fails, it can be transparently restarted: any ongoing stage executions can complete successfully, but a stage that completed before the Stage Tracker was restarted may be marked as having failed and may then be re-tried.

In this manner, the Stage Tracker robustly supports multiple concurrent task instances.

MedBox. On the smartphone, stages executed in a sandboxed environment called MedBox. MedBox exports capabilities necessary for crowd-sensing, and monitors resource usage by stages.

MedBox provides libraries that allow stages to access sensors, or that provide functions to manipulate sensitive data, as well as to transfer sensor data. Moreover, it also provides a library to access restricted storage on the phone where sensor data is stored. Stages can query this storage system using metadata attributes like location and time, allowing them to extract previously generated sensor data. Finally, MedBox supports triggers for stage execution, including triggers based on time or location. For example, a worker may be notified that she is near the location of an “Occupy” site. Notifications of these triggers may be delivered to users using any of the methods available in the underlying operating system. Stages are restricted by cloud-side static analysis to only access these libraries (as described above).

Finally, Medusa allows smartphone owners to specify limits on usage of system components such as CPU, network, or memory on

a stage or task instance. MedBox tracks the usage of each stage and aborts the stage if it exceeds its allocated resources. In the future, we envision extending the expressivity of these policies to permit privacy controls (e.g., opt-out for certain kinds of uploads) or reward-related stage-specific limits (e.g., tighter limits for stages where the monetary incentives are lower).

System Components	Requirements
MedScript Interpreter	Timeliness enforcement, robustness
Task Tracker	Supports multiple users, un-synchronized user operation, timeliness enforcement, robustness
Worker Manager	Incentives, crowd-curation, anonymity, security, worker-mediation
Stage Library	In-network processing, sandboxing, extensibility
Stage Tracker	Multiple concurrent stages, robustness, worker-mediation, privacy
MedusaBox	Stateless execution, sandboxing, concurrent stages, resource monitoring, access to stored data

Table 2: Medusa System Components and Properties

Summary. Medusa’s various components together satisfy the complex web of crowd-sensing requirements discussed in Section 2. Table 2 summarizes which components satisfy which requirements; a requirement may be satisfied by multiple components.

5. EVALUATION

In this section, we demonstrate four properties of Medusa: the expressivity of MedScript, the ability of the Medusa runtime to support concurrent task instance execution, the scalability of the runtime, and the robustness of Medusa.

Our experiments are conducted on a prototype of Medusa. The MedScript interpreter and Task Tracker are written using Python. Apache and PHP are used for accessing the MySQL based Stage State Table, as well as Stage Library. Worker Manager runs Tomcat with a Java Servlet environment and accesses AMT through a Java SDK². Finally, the Stage Library uses the `dex2jar` utility to convert Dalvik bytecode to Java, and the `javap` program from the Sun JDK for static program analysis.

Our smartphone software runs on Android v2.3.6. Stage Tracker is implemented as an Android service, and intercepts SMS and MMS messages³, and uses secure-HTTP based connections to the Stage Library and the Task Tracker. MedBox is implemented as a collection of Java threads.

To evaluate our system, we use commodity hardware both for the server and client side. In all our experiments, workers use either the Google Nexus One or the Nexus S. On the server, our cloud-based service is emulated by a Dell PowerEdge 2950, which has a dual-core Intel Xeon E5405 2.00GHz processor and 6MB built-in cache. Finally, we use data service from two GSM carriers in the United States to evaluate our push mechanism via SMS messaging.

5.1 Language Expressivity

To demonstrate the expressivity of MedScript, we have implemented ten qualitatively different crowd-sensing tasks (Table 3). Each task demonstrates different facets of Medusa, ranging from the use of different sensors to the use of different facilities provided by MedScript. Some have been proposed for participatory

²<http://mturksdk-java.sourceforge.net/>

³Our notifications may be longer than that allowed by SMS. Some carriers break up long initiation XML messages into multiple SMSs, others convert the SMS into an MMS.

Application	LOC*	Sensors	Properties
Video Documentation	86	Camera(Video)	In-network processing, Crowd curation
Collaborative Learning	62	Accelerometer, Audio	Different sensors
Auditioning	86	Camera(Video)	Reverse incentive mechanism, Crowd curation
Forensic Analysis	86	GPS, Audio	Access to stored data
Spot Reporter	45	Camera(Video), Audio	Text/Voice tagging
Road Monitoring	90	Accelerometer	Fork-join construct, App from PRISM [7]
Citizen Journalist	45	GPS	Multiple triggers, App from PRISM [7]
Party Thermometer	62	GPS, Audio	Fork constraint, App from PRISM [7]
WiFi and Bluetooth Scanner	45	Network Sensors	Apps from AnonymSense [6]

Table 3: Implemented Crowd-Sensing Apps. (*Line Of Code)

sensing in related work: we have implemented these in Medusa to demonstrate the power of the system.

Novel Crowd-Sensing Tasks. Many of the tasks in Table 3 are novel and are enabled by Medusa.

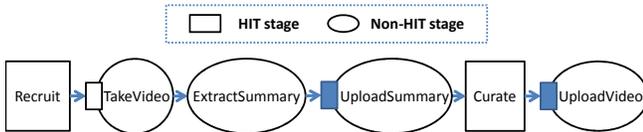


Figure 3: Video Documentation

Video Documentation. Although we have explained the code for the video documentation task in Listing 1, we discuss a few of the subtleties of this task here. First, for this and for other tasks discussed below, we illustrate the task program as a control flow diagram shown in Figure 3 (lack of space precludes a full program listing for the other tasks discussed below). HIT stages are represented by squares and SPC stages by ovals. Only the success connectors are shown, and the failure connectors are limited for brevity.

This task demonstrates some of the Medusa’s key features: in-network processing for extracting video summaries, crowd-curation, and opt-in data transfers (both when summaries are uploaded as well as when full videos are uploaded). Some stages are worker-initiated (e.g. `TakeVideo`), while others are automatically triggered when the previous stage completes (e.g., `ExtractSummary`). Worker-initiated stages have an unfilled rectangle before them, automatically triggered stages do not. A user may take multiple videos, so that multiple summaries can be uploaded for curation. Each upload stage requires worker opt-in (denoted by the filled rectangle before the stage). Only a subset of these summaries may be selected; the `Curate` stage returns a bitmask, which is used to determine the set of full videos that need to be uploaded. Finally, this task has multiple data upload stages: MedScript does not restrict uploads to the end of the control flow.

Collaborative Learning. The Collaborative Learning (Figure 4) task allows, for example, software vendors to solicit training samples for constructing robust machine learning classifiers (Section 2). Compared with video documentation, this task uses a different set of sensors (either an accelerometer or a microphone) and a dif-

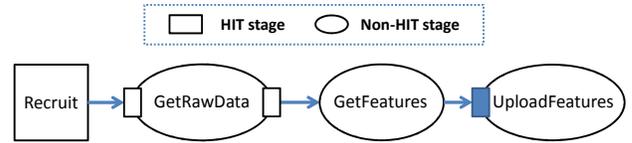


Figure 4: Collaborative Learning

ferent in-network processing algorithm (feature extraction). This task requires *annotation*, denoted by an unfilled rectangle *after* the `GetRawData` stage: workers are required to label the collected samples (e.g., as a sample of “walking”, or “running” etc.). If the worker collects multiple samples, she has to label each sample.

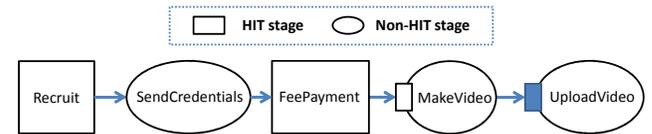


Figure 5: Auditioning (Full listing in Appendix A.)

Auditioning. Our auditioning task allows, for example, TV show producers to solicit videos from aspiring actors (Figure 5). The novel feature of this task is the support for reverse incentives: workers are required to *pay* the requestor for the privilege of submitting a video for auditioning. Reverse incentives are implemented by two stages. First, `SendCredentials` is an SPC stage that sends the worker’s AMT credentials (stored on the phone) to the Medusa cloud runtime. This step is necessary to set up the payment, which is implemented as a HIT stage `FeePayment`. This stage is conceptually similar to `Recruit`, except that the Medusa runtime sets up AMT as explained in Section 4. Once the requestor accepts and completes this task, payment is transferred; the worker can subsequently take a video and upload it. Uploaded videos are sent to the requestor (the TV producers in our example); we assume that selected participants are notified out-of-band (e.g., invited to an on-site auditioning).

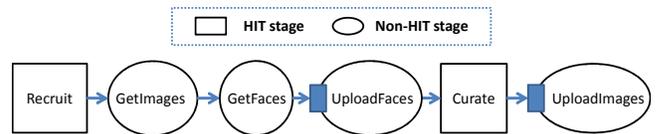


Figure 6: Forensic Analysis

Forensic Analysis. Our forensic analysis task permits security officials to obtain images of people who were at a specified location at a specified time when an incident occurred (Figure 6). The novel aspect of this task is access to historical sensor data stored on the smartphone. This access is accomplished through the `GetImages` stage, which, given a spatio-temporal predicate, obtains images whose meta-data matches the specified predicate. The subsequent `GetFaces` stage extracts faces from each image. Only the extracted faces are uploaded for curation. A security officer might visually inspect the uploaded faces to select faces corresponding to

potential suspects. The images corresponding to the selected faces are then uploaded.

Previously-Proposed Sensing Applications. Prior work has explored several participatory sensing applications, and Medusa can be used to program these. We have implemented these applications as crowd-sensing tasks, each of which begins with a `Recruit` stage using which workers can sign up for the task.

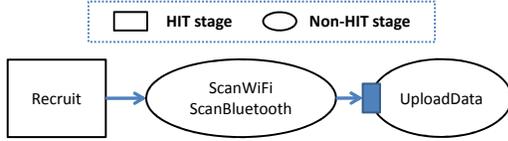


Figure 7: WiFi/Bluetooth Scanner

WiFi/Bluetooth Scanner [6] enables requestors to collect Wifi and Bluetooth neighborhoods from workers. This is implemented using a `netstats` stage, which models this network neighborhood information as outputs of a sensor. Requestors can use other stages to preprocess (e.g., filter) the data before upload.

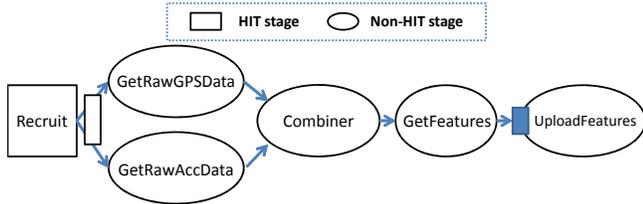


Figure 8: Road-Bump Monitoring

Road-Bump Monitoring [15], can be implemented using the task shown in Figure 8. This task collects GPS and accelerometer data, correlates data from the two sensors, and extracts features that are used to determine the location of bumps on roads. Its novelty is the use of the *fork-join* construct, which allows independent stages to concurrently sample GPS and the accelerometer.

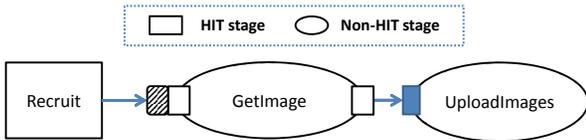


Figure 9: Citizen Journalist (Full listing in Appendix A.)

Citizen Journalist discussed in [10], asks workers to take an image at a specified location and label the image before uploading. The corresponding crowd-sensing task, shown in Figure 9, is unique in that one of the stages supports *two* triggers for initiating the stage: a location-based trigger which notifies the worker (shown as a hashed rectangle), who is then expected to initiate the taking of a photo.

Finally, we have also implemented a crowd-sensing task for the Party Thermometer application [7]. This task uses a location trigger to autonomously collect ambient sound, generate feature vectors, and upload them (Figure 10).

Quantifying Expressivity. Our Medusa task descriptions are very compact (Table 3): no task exceeds 100 lines. We also use code

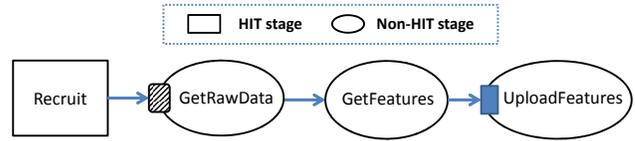


Figure 10: Party Thermometer

	Medusa	Standalone
Video Documentation	86	8,763
Collaborative Learning	62	7,076
Spot Reporter	45	17,238

Table 4: Line of Code Comparison

size to compare the complexity of three crowd-sensing tasks with standalone implementations of these tasks. Our comparison is only approximate because the implementations being compared are not precisely functionally equivalent.

Our work on Medusa has been inspired by our earlier standalone prototypes for video documentation and collaborative learning. The earlier prototypes did not include support for worker-mediation or incentives but had additional user interface elements for software configuration and management. In addition, a research group with whom we collaborate has implemented a spot reporter application, which tasks human agents to send audio, video, or text reports from the field. Our implementation of this task requires three stages, one to recruit agents, another to take the video (together with annotation) and the third to upload the result.

Table 4 quantifies the difference in code complexity using lines of code as a metric. Even given the differences in implementations, Medusa versions are about two orders of magnitude more compact than the standalone versions! Medusa’s runtime implements many functions common to these standalone applications, thereby reducing the programmer’s burden.

5.2 Concurrent Task Instances

The dynamics of task execution in Medusa can be quite complex, since computational and sensing stages can be interleaved with worker mediation. Thus, execution dynamics can be strongly influenced by human actions. At the same time, Medusa task throughput can be enhanced by its support for concurrent task instance execution.

To illustrate some of these properties of Medusa, we conducted an experiment where we instantiated one task instance for each of the ten prototype applications discussed above. Then, we asked four volunteers to sign up for a subset of the task instances. Each volunteer was given a phone and assigned an AMT worker account, but were otherwise unconstrained in how and when to complete the tasks.

Figure 11 depicts a timeline of the progress of the experiment. All task instances were completed within about 30 min. by the four workers. The upper panel shows the completion time of the task instances: these range from 6 min. to 30 min., with most instances completing within about 10 mins. The lone outlier is Citizen Journalist, which was location-triggered and could only be initiated when the worker had reached the specified location.

The timeline also depicts several interesting features of Medusa. Task instances can have differing numbers of SPC stages, indicated by the vertical bars along each timeline. Workers can concurrently sign up for multiple task instances; for example, worker I executes three task instances simultaneously. Workers sign up for tasks at

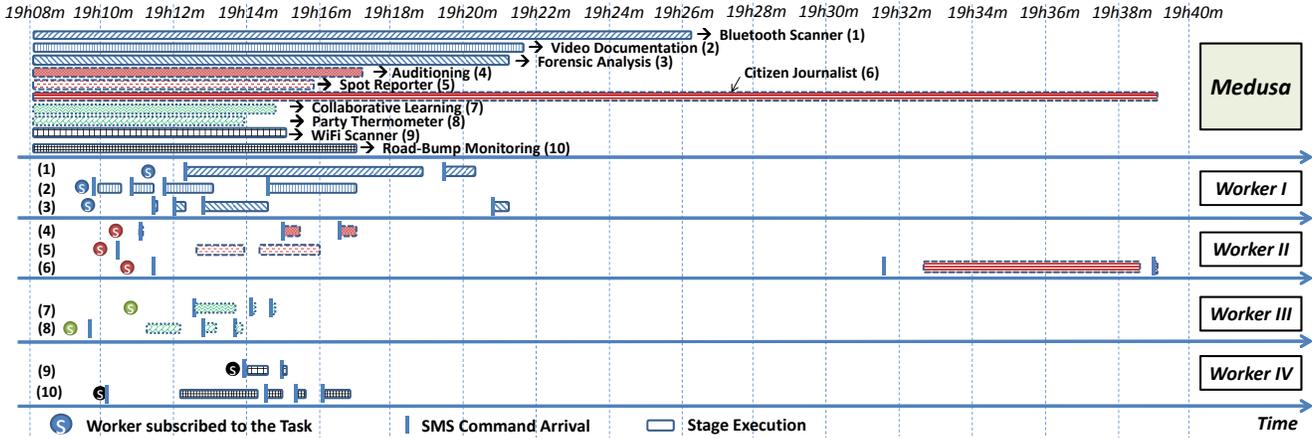


Figure 11: Concurrent Execution of Multiple MedScript Programs.

Component	Avg	Max	Min
Task Interpretation and Type Checking	26.64	26.89	26.45
Task Tracker Initiation Delay	0.84	0.90	0.77
Task Tracker latency amongst stages	0.87	0.94	0.83
[HIT] Delay upon the task registration request	4.83	12.03	3.61
[HIT] Waiting time for the registration response	2177.99	2824.52	1858.45
[HIT] Task execution delay	31039.83	50060.72	20028.91
[SPC] Delay on the request to AMT for commanding workers	1.30	2.56	0.77
[SPC] Messaging request confirmation from AMT	732.5	833.54	700.61
[SPC] Delay on SMS/MMS command message delivery	27000	78000	17000
[SPC] Task execution delay	3038.54	7039.54	1029.20
Total processing time	34.47	43.31	32.42
Total waiting time	63988.86	138758.3	40617.16
Total execution time	64023.33	138801.6	40649.58

Table 5: Delay break-down on the server (Unit: msec)

different times (between 1 min and 6 min into the experiment). Moreover, stage execution times for different tasks vary widely. For example, Video Documentation and Bluetooth Scanner have much longer stages than Spot Reporter. For some stages, stage execution commences immediately after Stage Tracker receives a command from Task Tracker; for others, waiting for worker initiation delays stage execution. Medusa seamlessly handles all these execution dynamics in a manner transparent to the task requestor.

5.3 Scalability and Overhead

To quantify the scalability and overhead, we instrumented our prototype to measure the time taken to perform several individual steps involved in task instance execution, both on the cloud runtime and the phone runtime. We then executed a single task instance on a Nexus One phone, where the task consisted of two stages: a *Recruit* (which requires human-mediation) followed by a null SPC stage (i.e., one that returns immediately and requires no worker-mediation). We present results from 10 trials.

Medusa Server Overhead. Table 5 shows server-side delays. Task

Component	Avg	Max	Min
Retrieve and parse SMS/MMS command message	38.2	58	26
Stage binary initiation time	402.7	506	374
Stage runner latency	6.2	14	3
Stage termination latency	12.7	14	3
Total overhead imposed	459.8	598	409

Table 6: Delay break-down on the phone (Unit: msec)

interpretation and type checking together take 26.64ms on average, Task Tracker initialization takes much less, 0.84ms. For the *Recruit* stage, a HIT request requires only 4.83ms; however, the time to register the task with AMT and receive a response requires 2.18s. To run an SPC stage, Task Tracker must prepare an XML command and send it to Worker Manager so that it can send a request for AMT to send SMS/MMS messages to the phone. Those two steps together incur 732.5ms of delay.

The largest component of delay in our system is the time that it takes to deliver an SMS message to the phone: between 20 to 50 *seconds*. This is because our implementation uses email-to-SMS gateway-ing; a more optimized implementation might use direct SMS/MMS delivery, reducing latency. That said, latency of task execution is not a significant concern for the kinds of tasks we consider: with humans in the loop, we anticipate crowd-sensing tasks to have deadlines on the order of hours from initiation. Indeed, the second largest component of delay in our system is the time that it takes for a human to sign up for the task.

The SPC stage execution time, measured from the perspective of the cloud runtime as the time from when the Task Tracker received confirmation message delivery from Worker Manager, to the time stage detection is completed is about 3 seconds. A large component of this latency is network delay, but another contributor is the polling period in the Task Tracker. Recall that Task Tracker polls the Stage State Table to determine if a state has completed. This polling time adds to the latency; in the future, we plan to alter our system so that Stage Tracker directly notifies Task Tracker of stage completion, which then saves state in the table, eliminating this latency.

Overall, we are encouraged by these results. The dominant delays in the system arise from notification using SMS and from waiting for human input. Actual processing overheads are only 34.47ms on average, which permit a throughput of 1740 task instances per minute on a single server. Moreover, this component is highly par-

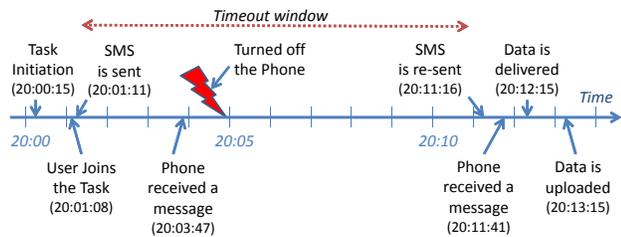


Figure 12: Failure recovery: turn off the phone in the middle.

allelizable, since each task instance is relatively independent and can be assigned a separate Task Tracker. Thus, if task throughput ever becomes an issue, it would be possible to leverage the cloud’s elastic computing resources to scale the system.

Medusa Runtime Overhead on the Phone. Table 6 presents the break-down of delay on the Medusa runtime on a Nexus One phone. Retrieving a stage initiation message from Task Tracker involves merging multiple SMS messages and parsing the resulting XML data object. This takes 38.2ms of delay on average. The major component of delay on the phone is the time to instantiate the stage implementation binary (in our experiments, the stage implementation was cached on the phone), a step which takes 402.7ms. This overhead is imposed by Android; our stage implementations are Android package files and the Android runtime must unpack this and load the class files into the virtual machine. Stage Tracker then performs some bookkeeping functions before invoking the stage, which takes 6.2ms. When the stage finishes its execution, Stage Tracker performs additional bookkeeping before notifying the cloud runtime, and this takes 12.7ms. The total delay imposed by Medusa platform is only 459.8ms per stage, of which the dominant component is the stage binary initialization. On the timescale of expected task completion (e.g., several 10s of minutes in our experiments above), this represents negligible overhead. Furthermore, we are aware of methods such as binary alignment which will allow us to optimize stage loading, but have left it for future work.

5.4 Robustness

Finally, we briefly demonstrate Medusa’s robustness.

Failure Recovery. The Medusa runtime maintains internal timeouts to recover from transient errors caused by messaging failure, battery exhaustion, or user actions (e.g., accidentally turning off the phone). To demonstrate failure recovery, we conducted an experiment where we turned off the phone during the scanning stage of the WiFi scanner task. Then we turned on the phone again after a couple of minutes. Figure 12 shows the sequence of events. Medusa waits for a pre-defined timeout (10 minutes in our implementation) before restarting the scanner stage. This stage then completes execution, eventually resulting in task completion. This simple retry mechanism works well across any transient failure mainly because stage execution state is maintained on the cloud.

Static Analysis on Stage Binary. To demonstrate the efficacy of using static analysis for sandboxing, we recruited three volunteers (none are authors of the paper) to add malicious code to existing stage implementations. We then tested whether our analysis was able to deter such attacks, which included opening HTTP connections, invoking sleep operations, accessing the SD card, and so forth. Of the nine modifications made by the volunteers, our static analyzer caught 7 correctly (Table 7). The two attacks that it did not catch targeted resource usage directly: infinite recursion,

Code Modification	Result
Make one function sleep for a long time	REJECT
Write a file to SD card	REJECT
Time string format translation	REJECT
Open HTTP connection	REJECT
Delete all files in SD card	REJECT
Vector operation	ACCEPT
Throws exceptions	REJECT
System.exit(-1)	REJECT
Recursive calls	ACCEPT
Fills the heap until memory is exhausted	ACCEPT

Table 7: The defense capability of static analyzer against arbitrary code modification.

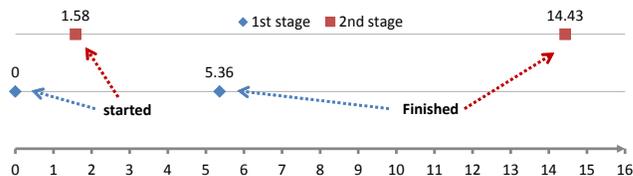


Figure 13: Robustness: forcing a limit on computation.

and attempts to exhaust the heap. These attacks require dynamic tracking: Medusa also has a dynamic resource tracking component, which we demonstrate next.

Limiting Resource Usage. Medusa allows smartphone owners to place limits on the computation used by a single stage (on a per-stage basis) and the amount of data transferred per task instance. Using these controls, users can implement policies which, for example, apportion more resources to tasks with higher monetary reward.

To illustrate how these limits are enforced, we created a single long-running stage, and concurrently ran two instances of that stage with limits of 5 and 10 seconds respectively. MedBox maintains a watchdog timer, which, in our implementation, checks stage resource usage every three seconds. As shown in Figure 13, it successfully terminated our stages at 5.36 seconds and 12.85 seconds; given the granularity of our timer, stages may get bounded additional resources beyond their limits. This kind of mechanism can protect against CPU resource exhaustion attacks.

Medusa can also protect against excessive network usage per task instance. Since all data transfers must use MedBox’s data transfer library, that library can account for data transfers on a per task basis. We conducted an experiment in which a task instance attempted to upload 60 image files and was allocated a 3MB limit: MedBox successfully terminated the task (by notifying the Task Tracker) just before it reached the 3MB limit (the transfer library transfers data in 200KB chunks, and checks whether the limit would be exceeded before transferring data) as depicted in Figure 14.

6. RELATED WORK

To our knowledge, no other prior work describes a programming language and runtime system for crowd sensing. Crowd-sensing, as we have defined it, combines two strands of research: participatory sensing and crowd sourcing. We now describe how these strands relate to Medusa.

Participatory sensing systems like PEIR [16] and SoundSense

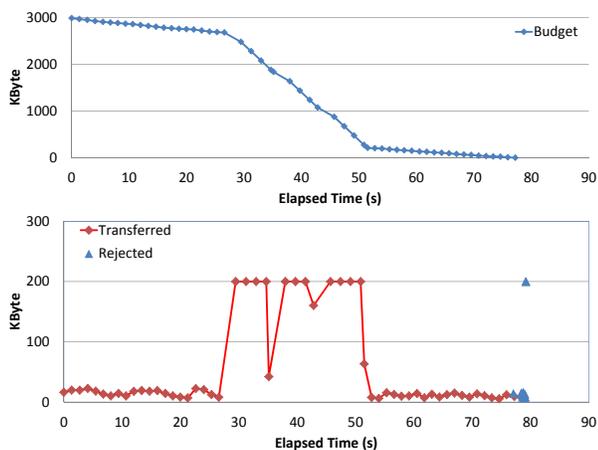


Figure 14: Failure recovery: harnessing network data usage.

[13] do not incorporate human-mediation or incentives into sensing. Moreover, they do not support programmable data collection from multiple sensors or an in-network processing library. Some pieces of work in this area, however, share some goals with Medusa. Campaignr [2] is an early effort on programming sensor data collection on a single phone for participatory sensing and uses an XML description language to specify data collection tasks and parameters.

AnonySense [6] is a privacy-aware tasking system for sensor data collection and in-network processing, while PRISM [7] proposes a procedural programming language for collecting sensor data from a large number of mobile phones. More recently, Ravindranath et al. [18] have explored tasking smartphones crowds and provide complex data processing primitives and profile-based compile time partitioning. Unlike these systems, Medusa incorporates incentives and reverse incentives, human-mediation, and support for curation into its programming framework.

Several systems have integrated worker-mediation into their workflow. Most of the systems have been inspired by, or directly employ, Amazon Mechanical Turk (AMT [1]). CrowdDB [9] extends relational databases to support human-mediation in SQL query workflows. CrowdSearch [22] exploits AMT to crowd-source video search. TurKit [12] enhances the original programming model of AMT by allowing repeated execution of existing tasks, and enhances the robustness of the AMT programming model. Additionally, several studies have explored who uses AMT and in what ways [11, 5]. Medusa is qualitatively different from this body of work in that it enables users to contribute processed sensor data, a capability that enables many interesting applications as described above.

Complementary to our work is PhoneGap [3], a system that enables developers to program mobile applications using HTML5 and JavaScript, then automatically generates platform-specific native binaries. Thus, PhoneGap focuses on single source multi-platform code development, and not on crowd-sensing.

Finally, many of our applications have been inspired by prior work. Conceptual descriptions were provided in [17] for video documentation and in [14, 4] for collaborative learning. However, these papers did not discuss incentives or worker-mediation. Road-bump monitoring was described in [15], and citizen journalist in [10], but the focus of research in these papers (sensing algorithms, energy, location discovery) was very different from ours.

7. CONCLUSIONS

Medusa is a programming system for crowd-sensing. The MedScript programming language provides high-level abstractions for stages in crowd-sensing tasks, and for control flow through the stages. Medusa supports specifying various forms of worker mediation in the sensing workflow. A runtime, partitioned between the cloud and smartphones, achieves a complex web of requirements ranging from support for incentives to user-specified controls on smartphone resource usage. Medusa task descriptions are concise, and the runtime system has low overhead.

Finally, our work suggests several future directions, including: gaining deployment experience with more crowd-sensing tasks; reducing the overhead of the Medusa runtime and the latency of its SMS-based notification; replacing the XML-based specification language with a visual paradigm using OpenBlocks [19] and AppInventor [21]; using Medusa a distributed programming framework that crowd-sources computations to mobile devices.

Acknowledgements

We would like to thank our shepherd, Landon Cox, and the anonymous referees, for their insightful suggestions for improving the technical content and presentation of the paper.

8. REFERENCES

- [1] Amazon mechanical turk, <https://www.mturk.com/>.
- [2] Campaignr:configurable mobile sensing, <http://urban.cens.ucla.edu/technology/campaignr/>.
- [3] Phonegap, <http://phonegap.com/>.
- [4] X. Bao and R. R. Choudhury. Movi: mobile phone based video highlights via collaborative sensing. In *Proc. ACM MOBISYS*, 2010.
- [5] L. Chilton, J. Horton, R. Miller, and S. Azenkot. Task search in a human computation market. In *Proc. HCOMP*, 2010.
- [6] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. Anonymsense: Privacyaware people-centric sensing. In *Proc. ACM MOBISYS*, 2008.
- [7] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma. Prism: Platform for remote sensing using smartphones. In *Proc. ACM MOBISYS*, 2010.
- [8] C. Dwork. Differential privacy. In *ICALP*, pages 1–12. Springer, 2006.
- [9] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *Proc. ACM SIGMOD*, 2011.
- [10] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. "Micro-Blog: Sharing and Querying Content Through Mobile Phones and Social Participation". In *Proc. ACM MOBISYS*, 2008.
- [11] G. Little, L. Chilton, M. Goldman, and R. Miller. Exploring iterative and parallel human computation processes. In *Proc. HCOMP*, 2010.
- [12] G. Little, L. Chilton, M. Goldman, and R. Miller. Turkit: Human computation algorithms on mechanical turk. In *Proc. ACM UIST*, 2010.
- [13] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *Proc. ACM MOBISYS*, 2009.
- [14] E. Miluzzo, C. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. Campbell. Darwin phones: the evolution of

sensing and inference on mobile phones. In *Proc. ACM MOBISYS*, 2010.

- [15] P. Mohan, V. N. Padmanabhan, and R. Ramjee. "Nericell: rich monitoring of road and traffic conditions using mobile smartphones". In *Proc. ACM SENSYS*, 2008.
- [16] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In *Proc. ACM MOBISYS*, 2009.
- [17] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. Energy-delay tradeoffs in smartphone applications. In *Proc. ACM MOBISYS*, 2010.
- [18] L. S. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones. In *Proc. HotMobile*, 2012.
- [19] R. V. Roque. OpenBlocks: an extendable framework for graphical block programming systems. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2007.
- [20] W. H. Whyte. The Social Life of Small Urban Spaces. Project for Urban Spaces, 1980.
- [21] D. Wolber. App inventor and real-world motivation. In *Proc. ACM SIGCSE*, 2011.
- [22] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: Exploiting crowds for accurate real-time image search on mobile phones. In *Proc. ACM MOBISYS*, 2010.

APPENDIX

A. MORE MEDSCRIPT PROGRAMS

A.1 Citizen Journalist App

```

1 <xml>
2 <app>
3 <name>Citizen-Journalist </name>
4 <rrid>[User's Requestor ID]</rrid>
5 <rrkey>[User's Requestor Key]</rrkey>
6
7 <stage>
8 <name>Hiring </name> <type>HIT </type>
9 <binary>recruit </binary>
10 <config>
11 <stmt>Citizen Journalist Demonstration </stmt>
12 <expiration>>21:00:00 12/16/2011 </expiration>
13 <reward>.05 </reward>
14 <output>W_WID</output>
15 </config>
16 </stage>
17 <stage>
18 <name>TakePicture </name> <type>SPC </type>
19 <binary>mediagen </binary>
20 <trigger>location=34.2523391 - 118.277907140, user-initiated </trigger>
21 <review>textdesc </review>
22 <config>
23 <params>-t image </params>
24 <output>IMAGE</output>
25 </config>
26 </stage>
27 <stage>
28 <name>UploadData </name> <type>SPC </type>
29 <binary>uploaddata </binary>
30 <trigger>none </trigger>
31 <config>
32 <input>IMAGE</input>
33 </config>
34 </stage>
35
36 <connector>
37 <src>Hiring </src>
38 <dst> <success>TakePicture </success> <failure>Hiring </failure> </dst>
39 </connector>
40 <connector>
41 <src>TakePicture </src>
42 <dst> <success>UploadData </success> <failure>Hiring </failure> </dst>
43 </connector>
44 </app>
45 </xml>

```

Listing 2: Citizen Journalist

A.2 Auditioning App

```

1 <xml>
2 <app>
3 <name>Auditioning </name>
4 <rrid>[User's Requestor ID]</rrid>
5 <rrkey>[User's Requestor Key]</rrkey>
6 <rwid>[User's Worker ID]</rwid>
7 <deadline>>21:00:00 12/16/2011 </deadline>
8
9 <stage>
10 <name>Recruit </name> <type>HIT </type>
11 <binary>recruit </binary>
12 <config>
13 <stmt>Recruiting for Audition App. </stmt>
14 <expiration>>21:00:00 12/16/2011 </expiration>
15 <reward>.05 </reward>
16 <output>W_WID</output>
17 </config>
18 </stage>
19 <stage>
20 <name>SyncENV </name> <type>SPC </type>
21 <binary>helloworld </binary>
22 <trigger>none </trigger>
23 </stage>
24 <stage>
25 <name>FeePayment </name> <type>HIT </type>
26 <rid>W_RID</rid> <rkey>W_RKEY</rkey> <wid>R_WID</wid>
27 <binary>recruit </binary>
28 <config>
29 <stmt>Audition Fee Payment.(100 dollars) </stmt>
30 <expiration>>21:00:00 12/16/2011 </expiration>
31 <reward>100 </reward>
32 </config>
33 </stage>
34 <stage>
35 <name>MakeVideo </name> <type>SPC </type>
36 <binary>mediagen </binary>
37 <trigger>user-initiated </trigger>
38 <config>
39 <params>-t video </params>
40 <output>VIDEO</output>
41 </config>
42 </stage>
43 <stage>
44 <name>UploadVideo </name> <type>SPC </type>
45 <binary>uploaddata </binary>
46 <trigger>user-initiated </trigger> <review>yesno </review>
47 <config>
48 <input>VIDEO</input>
49 </config>
50 </stage>
51 <stage>
52 <name>Evaluation </name> <type>HIT </type>
53 <rid>W_RID</rid> <rkey>W_RKEY</rkey> <wid>R_WID</wid>
54 <binary>vote </binary>
55 <config>
56 <stmt>Evaluation Press. If you like the video, press Yes. Otherwise No
57 </stmt>
58 <expiration>>21:00:00 12/16/2011 </expiration>
59 <reward>.01 </reward>
60 <numusers>1 </numusers>
61 <input>VIDEO</input>
62 <output>RESULT</output>
63 </config>
64 </stage>
65 <connector>
66 <src>Recruit </src>
67 <dst>
68 <success>SyncENV </success>
69 <failure>Recruit </failure>
70 </dst>
71 </connector>
72 <connector>
73 <src>SyncENV </src>
74 <dst>
75 <success>FeePayment </success>
76 <failure>Recruit </failure>
77 </dst>
78 </connector>
79 <connector>
80 <src>FeePayment </src>
81 <dst>
82 <success>MakeVideo </success>
83 <failure>Recruit </failure>
84 </dst>
85 </connector>
86 <connector>
87 <src>MakeVideo </src>
88 <dst>
89 <success>UploadVideo </success>
90 <failure>UploadVideo </failure>
91 </dst>
92 </connector>
93 <connector>
94 <src>UploadVideo </src>
95 <dst>
96 <success>Evaluation </success>
97 <failure>Recruit </failure>
98 </dst>
99 </connector>
100 </app>
101 </xml>

```

Listing 3: Auditioning