Campaignr
A Framework for Participatory Data Collection on Mobile Phones

No Author Given

No Institute Given

Abstract. Participatory sensing takes advantage of the pervasive nature of mobile phones to collect data about the urban environment using the available sensors. Campaignr makes collecting this data as simple as a few button pushes. It provides access to the sensors in a robust and flexible way that hides the complexities of the mobile embedded phone environment. This paper describes the design choices and provides some numerical evaluation of Campaignr. Campaignr has been and is being actively used as the data collection method for many research projects, both internally and externally.

1 Introduction

Data collection for the urban environment is difficult. The user, although a voluntary participant, does not want to be encumbered by heavy, expensive, special purpose equipment that constantly gets in the way of normal behavior. Additionally, high variability and noise makes the use of static sensors difficult. Scientists interested in being able to examine specific information generated by users have previously had to expend significant effort creating applications and/or designing hardware that are used only a handful of times at most, and are often difficult to modify when new types of data are required[1]. What is needed is an application that requires no programming for the party interested in the data, is flexible enough to change functionality with minimal effort, is robust enough to handle non ideal conditions, and supports a wide range of sensing modalities.

This paper presents Campaignr, a software framework for mobile phones that enables owners of smartphones (specifically Symbian S60 3rd Edition phones) to participate in data gathering campaigns. Campaigns are set up by individuals or groups of people that are interested in exploring a specific piece of the urban environment[2]; both social[3] and environmental[4] concerns. Campaignr can upload the data collected by a smart mobile phone to any online storage that supports a specific XML format, or can store the information on the phone's memory for later retrieval when internet connectivity is unavailable. Because Campaignr is designed to work without a connection it can support data collections in areas of poor or no connectivity, enabling collection in more rural or wilderness settings. Campaignr provides easy access to the hardware sensors such as camera, microphone, cell tower information, and GPS (both internal and external via bluetooth). It also can access metadata that provides relevant and useful information, such as the current time of the phone, the globally unique serial number of the phone, and user generated text input. A campaign is defined by an XML file that contains options on which sensors to collect, how to collect data from those sensors, where to put the collected data, and how the user interface behaves. For example an author of a campaign could have the user push a button on the phone to have Campaignr take a picture that the user can frame in a viewfinder, and attach the time and location coordinates the picture was taken and then upload it to SensorBase[5]. Or, with just a few quick changes of the XML, the author can have Campaignr automatically take the picture every minute

without involving the user after the campaign has been started on the phone. No changes to the underlying code are needed. No recompiling, testing, debugging, or reinstalling is needed, except for putting the modified XML campaign file back on the phone. Development is significantly harder than for desktop applications as the smartphone is an embedded environment with its own language, run time environment, and a more involved process of getting the code successfully installed onto the phones[6].

The next section expands on why Campaignr was written from the ground up, rather than making use of preexisting technology, the third section explains how Campaignr works, the fourth section discusses related work, the fifth section presents data relating to the operation of Campaignr, and the sixth section discusses the direction that Campaignr is headed.

2 Motivation

The state of the art when it comes to using mobile phones for taking measurements of the environment, be it urban or rural is largely undeveloped. Static, limited-configurability, and small number of sensing modalities is the norm. This often makes the applications essentially "one-offs" because it is easier to create another application to suit the needs of the new sensing objective than modify the old application. Which leads to reimplemented code, reimplemented ideas, and wasted time spent creating applications from scratch.

There are several programming languages available for writing software for mobile phones. However those languages are limited in both features and performance. J2ME[1] is a version of Java specifically designed for resource-constrained devices such as mobile phones. But J2ME is designed for what are called "feature phones", mobile phones that have very limited hardware and software (i.e. not smartphones). Since the common base is so low, the more advanced features needed by richer data collection campaigns are either implemented in optional APIs that are not supported in all phones or poorly implemented, or are not supported by any part of J2ME at all. One example of a feature that is useful to have and is a basic part of all GSM phones, is the ID of the cell tower that the phone is currently connected to. It is currently impossible to obtain the cell tower ID in J2ME without having to write custom Symbian C++ and socket code. The efficiency of J2ME is, as well, inherently lower than natively compiled Symbian code; the Java byte code is interpreted and then run through the JVM. J2ME does provide the advantage of being able to be run across multiple operating systems to reach a wider number of phones. However, trying to make a complex application work using the core provided J2ME APIs may not be possible.

Python is another language that is able to leverage previous experience on the mobile embedded device platform. However it is not as feature rich nor as efficient as native Symbian C++ code. Python, as implemented on Symbian (and on the desktop), is a wrapper for calls to the native code base. That makes it much easier to write code and provides more functionality than does J2ME. However everything in Python for S60 phones[7] is implemented directly on top of Symbian classes. One example is base-64 encoding. The function is implemented in pure Python and will take a longer amount of time. However the learning curve for Python is lower and provides a simpler programming model. Taking a picture with the camera takes 3 lines of Python code, but takes multiple large classes and lots of testing and debugging in Symbian C++ to provide the same functionality.

Gathering information about the urban environment, be it detailed location traces to use in pollution exposure assessment, or hazardous sidewalks, or food eaten, holds transformative for many fields of research from urban planning to medicine. So far there have been no easy

[1] Java 2 Platform, Micro Edition

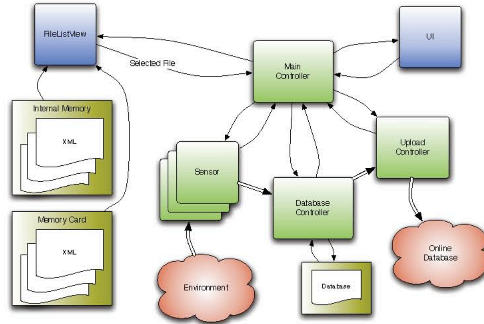

Fig. 1. The flow control diagram for Campaignr. Single line arrows are control lines and double line arrows are data lines. Campaignr starts with reading the XML files from memory and displaying a list of them. Then once one is selected, the main controller will create all the other parts based on what is in the XML. Data is then collected from the sensors, stored in the database, and then uploaded over WiFi or EDGE.

ways to take advantage of the commodity sensors and communication devices that people carry around with them all day, every day. These small, well packaged hardware devices have already proved invaluable in information exchange. The basic mobile phone inherently provides two sensing modalities and a means to extract readings. The microphone can record audio, the cell tower can provide coarse location, and the radio can send out information. Many modern mobile phones also provide one more sensing modality by including a camera. In addition to the hardware sensors, there are also metadata that are easy to collect such as the current time and the International Mobile Equipment Identity[2] (IMEI) number of the mobile phone. It is also possible to prompt the current user of the mobile phone to provide information, be it text that they manually enter or a choice from a list of options. Mobile phones will continue to become more sophisticated providing more reasons and options with which to collect data.

Campaignr has to provide access to all the possible sensors and has to work well in challenging network environments without needlessly exposing the user to the complexities of mobile phone programming or endlessly pestering the user for input or direction. People who have had no programming experience should also be able to take the same advantage of the sensing capabilities offered by a mobile phone as those versed in mobile phone programming. Campaignr is completely controlled by a campaign XML file that tells it what sensors to use, how to capture the data, and where and how to upload the data. The XML file can also control the look and function. XML was chosen because it is easy for a human to read and create or modify and yet also easy for a machine to generate and parse. XML's usability has already proven itself on the web.

3 Implementation

Campaignr is written in Symbian C++ for the Symbian Operating System, which predates the ANSI C++ standard. This helps, in part, to explain why Symbian C++ is hard to pick up,

[2] The IMEI number is globally unique amongst all mobile phones. No two phones can have the same number, making it a useful way to distinguish phones.

```
Sample.xml
    <campaign name="Sample" startOnLoad="true">
      <hide_status/>
      <hide_options/>
      <automatic>
5:      <sensor type="image">
          <size>1280:960</size>
        </sensor>
        <sensor type="timestamp"/>
        <sensor type="imei" name="id">
10:     <upload type="sensorbase.org">
          <project id="1">
            <table name="meal_capture">
              <field name="image" sensor="image"/>
              <field name="timestamp" sensor="timestamp"/>
15:           <field name="user" sensor="id"/>
            </table>
          </project>
        </upload>
        <interval>10</interval>
20:   </automatic>
    </campaign>
```

Fig. 2. Sample XML campaign file that is used to task Campaignr. In this case, an image is collected every 10 seconds along with the current phone time and the unique phone serial number. Then the data are uploaded to the meal_capture table in SensorBase. The campaign will start right away after being chosen and the top section will be hidden and the Options menu will be disabled as seen in Figure 3.

even with knowledge of ANSI C++. There are many notable differences that make needing to learn Symbian C++ for creating a campaign highly undesirable.

Instead of `throw` and `catch`[3] there is `TRAP` and `User::Leave()`[23]. The `TRAP` and `TRAPD` macros carry other baggage that is otherwise automatically handled in ANSI C++. Since the error handling code is not part of the syntax of C++ there is no way for the compiler to check that *leaves* are handled properly; consequently it is up to the programmer. The programmer has to push and pop heap objects on and off of a `CleanupStack` in case a function call *leaves* and there are still open handles or heap memory in use. The user has to explicitly keep track of all pieces of data in case a function call *leaves*, although it may not. Yet the cleanup stack does not work for all possible cases. It is possible that instantiating a class could cause a *leave* if the phone were to run out of memory and if the constructor itself put something on the heap that would cause a memory leak, since Symbian can not clean up the uninitialized class itself. Therefore class instance construction is split into two stages. The first stage is the standard C++ constructor, but that must not contain any code that could possibly *leave*. The second stage constructor called `ConstructL` is used to execute any setup code that could *leave*.

The use of threads is heavily discouraged due to the extra resources threads taken up and decreased efficiency caused[24]. Symbian OS provides a class that encapsulates a wait loop so that asynchronous calls are still supported but in a more efficient, yet confusing way.

[3] Which did not gain decent compiler support until after Symbian C++ was created.

Fig. 3. Screenshot of what Campaignr looks like while running the code from Figure 2

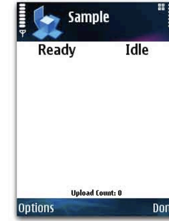

Fig. 4. Screenshot of what Campaignr looks like before starting the collection process and with the default UI settings.

All these factors together make it difficult to learn Symbian C++. At the same time there are many deployed and planned mobile phones that are Symbian based that should be leveraged. So instead of making the end user have to program in this difficult language, Campaignr was developed to hide the complexities of the underlying system from the users who should not have to learn or deal with its idiosyncrasies. Campaignr provides a simple interface and method of configuring that only takes minutes to learn. It is a testament to how easy Campaignr is to use that it has been heavily used by research groups already, both internally and externally.

The XML given by Figure 2 tells Campaignr to run the `Sample` campaign that will take an `image`, add a `timestamp` and the phone's `imei`, and upload it to a table called `meal_capture` every 10 seconds. The campaign will also start right away without the user having to manually start it, and the upper part of the display will be hidden and the `Options` menu disabled to prevent turning off the collection or upload processes as shown in Figure 3.

Campaignr can be thought of as consisting of five key components: the main controller (3.1), the sensors (3.2), the database (3.3), the upload controller (3.4), and the user interface (3.5), which are described below.

3.1 Main Controller

The main controller is the glue that connects everything together and contains the logic that makes Campaignr work. After the user selects a campaign from the list that is presented to the user upon Campaingr startup, the filename is passed to this controller which then parses the XML file to determine what sensors to instantiate, how to collect from them, what the database table(s) should look like (creating them if they do not already exist), and where and what to upload. There are also options in the XML for the user interface and overall campaign behavior. This controller knows how to instantiate each type of sensor and keeps track of all the ones needed for the given campaign. It then creates the database and upload controllers and passes the specified options to them. This controller controls the lifecycle of everything it creates, controls the flow of execution for data collection, storage, and upload, and handles the basic user interface as described in 3.5.

There are two ways in which to collect data from the sensors, manually or automatically. In manual mode Campaignr waits for the user to push the center directional pad button to begin a collection run. In automatic mode Campaignr will collect data from the specified sensors on the interval specified. Campaignr can also handle a manual and automatic mode in the same campaign.

After everything is instantiated and set up properly, this controller then waits for the user to start the data collection and upload processes. Or if so specified in the XML, the campaign will start as soon as Campaignr finishes loading. It iterates through one of its list of sensors depending on whether it is performing an automatic or manual capture and informs each sensor in turn to take a sample. This process is implemented asynchronously because a single sample from a sensor could potentially take a non negligible amount of time to collect and the software should not freeze and become unresponsive to the user at any point. Since this controller will not know when the current sensor will finish collecting its data, it waits for the sensor to notify it that the sensor has finished. The main controller passes a pointer to itself to the current sensor so that the sensor can call a method to inform the main controller when it has finished collecting its sample. When the sensor is done and notifies the controller, the main controller tells the next sensor in the queue to take its turn. And when that one is done, the next, and so on until all sensors in the list have had a chance to collect data. This process also has the benefit of allowing other processes to run, such as manipulating the database or uploading data, since the processor is not tied up with busy waiting.

Both an automatic set of sensors and a manual set of sensors can be specified in a single XML campaign file. They are maintained separately and go to different tables in the same database so that they do not get in each other's way. However it is only possible to collect from one set of sensors at a time. If one of the sets of sensors is currently collecting, the other set is unable to do its own collecting. Even if the automatic collection process is taking a longer time to go through than its interval is set for, Campaignr will still wait for the current one to finish. How Campaignr deals with this request for data while, already collecting data, differs depending on whether the request is for an automatic or for a manual data capture. Currently automatic requests are dropped because otherwise if they were queued up, that queue could become unbounded if the automatic interval is smaller than the amount of time it takes to collect data. It is up to the creator of the campaign to set a reasonable interval. The worst that could happen in Campaignr is that any automatic sample that occurs while one is already running will get dropped.

If there is an automatic collection process running and the user pushes a button to have a manual sample taken, then that request is queued up and will be handled as soon as the automatic one is done. If the manual request was ignored the user would not understand why pushing the button sometimes worked and sometimes did not, and that would lead to the user mistrusting the application. However if the person tries to take a sample while there already is a manual collection going on, it currently is dropped since the user has more context about what Campaignr is doing than in the automatic case and can easily determine the reason for the request being ignored.

The asynchrony is implemented using Symbian C++'s Active Objects[25]. Active Objects are implemented by inheriting from a special class that defines some instance variables and methods that are used to make and handle asynchronous calls. A subclass can make itself "active" to let the underlying framework know that it wants its request handling method to be called at some point in the future. That point comes after the class's status variable is set from pending to active. The variable can be set by the class itself when splitting up long running processing or it can be passed to another class that will then toggle the variable to let the calling class know it is done fulfilling the request. This way of doing asynchronous processing is preferred over threads since it saves stack space and no locking has to be implemented. But Active Objects are difficult to understand and the learning curve for Symbian programming. This is yet another complex issue that Campaignr is able to hide.

When all the sensors have finished their collections, and if uploading is enabled, the main controller tells the upload controller to attempt to upload the saved data. This process is described in more detail in section 3.4

3.2 Sensors

Campaignr started with just four sensors (two hardware and two metadata) when it was first released and has now grown to twelve sensors with more planned for the near future. The sensors cover a wide variety of possible inputs, from the onboard camera and microphone to time stamps and text input by the user. There are even sensors for discovering what bluetooth devices or wifi access points can be seen by the phone.

Sensors all follow the same process of collecting data. After they are instantiated, the main controller calls their `SetAttrsL`[4] method. This method is used to set the name of the sensor (since it is possible to have multiple of the same type of sensor sensed from multiple times in one data collection run and there needs to be a way to identify exactly which one is which) as well as set any sensor specific options that were specified in the XML file. The main controller does not try to parse sensor specific options, it passes the whole sensor tag to the corresponding sensor. For example the `size` tag on line 6 of Figure 2 is not seen by the main controller, but is instead parsed by the `image` sensor.

When a sensor is required to collect data, the main controller calls the sensor's `GetData` method, passing a pointer to itself and to the database controller. The sensor then gathers its specific data. This can be either an immediate value determined at set up time or by a background process or a value that could potentially require a non trivial amount of time to complete and therefore need to be executed asynchronously. In either case the sensor uses the database pointer to call its `AddValueL` method, passing the sensor's unique name and data. (The reason for passing the name is explained in section 3.3.) Then the sensor cleans up its temporary processing data (if any) and calls the main controller's[5] `GetDataDone` method if the data was successfully collected or the main controller's `GetDataError` method if something went wrong to alert the main controller to abort the current data collection run and not commit the new row to the database. The details of how the sensor works is sensor specific. The sensor can be as simple as passing a string, that was defined during setup, to the database and signal the main controller that it's done collecting data all within the `GetData` method so that the sensor doesn't need to be an Active Object. Or, on the other extreme, the sensor can do an asynchronous query of the MAC addresses of what bluetooth devices are within range of the mobile phone and also then query each for it's name and finally send that queried data to the database at some later point in time after the data has all been collected.

Campaignr can, without much difficulty, be extended to include more sensors as can be seen by the rapid increase to date. New sensors can be attached via bluetooth, infrared, or even the serial port or can be associated with new metadata. Symbian C++ classes can be created that implement the proper interfaces discussed above and only two files need to be modified to tell Campaignr about the new sensors. Creating the skeleton class, changing those two files and compiling takes just a few minutes. Actually implementing the logic of the sensor is the hard part. Currently all the sensors have to be predefined and compiled into Campaignr

[4] the `L` suffix is a Symbian C++ convention that means that the function might leave. Yet another detail that Campaignr is able to hide.

[5] Actually it is designed that it doesn't have to be the main controller, but anything that implements the proper mixin class. A mixin or M class is an abstract class that defines methods that the implementing class has to handle, essentially using C++'s multiple inheritance like Java's interfaces.

to work. But the goal, in the future, is to be able to create a plugable architecture for sensors so that anyone can create a sensor without needing to modify any files within Campaignr nor recompile. The new sensors could even be installed separately if the situation calls for it.

3.3 Database

The database controller is a wrapper around Symbian's proprietary database implementation to provide access and integrity control and to easily set up the properly formatted tables for the campaign. It makes sure that the data persists across multiple runs of the same campaign and that it is not lost when or if the battery complete drains, the network connectivity drops, or Campaignr unexpectedly quits. Campaignr was designed from the very beginning to be highly robust when it comes to dealing with the collected data. Data is written into the database before anything else. The only data that could be lost the data that is currently being sensed.

The database controller started off as a very simple wrapper with a few helper methods. But it quickly became obvious that the underlying database APIs could not handle arbitrary reading, writing and deleting from the database. The database entered into a bad state if rows were deleted while in the middle of adding one. Originally a table was opened and left open across multiple calls to the database controller while collecting data. But the database implementation did not handle that mode of access; it corrupted the database. So the controller was changed to store each sensor reading in memory until all sensors had had a chance to collect data, and then the data was added to the database with a single SQL insert statement. The database controller also currently implements row querying, row deletion, and arbitrary[6] SQL execution in an asynchronous manner so that queries can be queued up and not tie up the processor waiting for previous queries to complete nor interfere with each other. The database is heavily used by different competing classes that have no (and should not need to have) knowledge of each other and therefore the database protects itself from conditions that could cause data loss. The upload controller accesses the database frequently to upload rows and then delete those rows upon successful transmission, and if a row was being added at the same one of those queries happen, the data in the database becomes irretrievable.

Sensors add their data to the database by calling the database controller's `AddValueL` method. Each value is converted and temporarily stored as a string in an array while the rest of the sensors add their own values. Once all the sensors have finished the main controller commits the row by calling the `FinishAddingRowL` method which asynchronously generates the properly formated SQL insert statement and executes it. The proper statement requires the name of the columns in the table in case any of the values are missing. The proper column names are easy to discover since they are the unique names of the sensors. So to know which value maps to which sensor, a second array that contains the sensor names is also required. When a sensor adds its value to the row, it also passes its name to support the more flexible row insertion.

3.4 Upload Controller

The upload controller handles everything that pertains to sending the data to somewhere and removing the safely uploaded data. The controller handles connecting to the internet via GSM or WiFi (if the phone supports it), packaging up the data in a way the destination expects, handling responses from the destination, and then telling the database which rows

[6] Arbitrary as in SQL that other parts of Campaignr use, not arbitrary as in input from an unknown source.

to delete. There is currently only one format for uploading and that is XML formatted data and a few other credentials in the `application/x-www-form-urlencoded` MIME type. The upload controller first asks for a few rows of data from the database. If there are none, the controller does not continue. When the database has notified the upload controller that the data is ready, the upload controller extracts the data that has been specified to upload by the XML campaign file from each returned row and sticks it in its own XML tag, with the binary data base 64 encoded for transmission over HTTP POST. Then the generated XML, as a string, and the other credentials, such as where exactly the data should go and who is uploading the data, are put into a form that is then url encoded. After the form is ready, a connection to the internet is created, if one is not already active, and the data is POSTed to the url specified in the XML file or the default location if none is specified. The upload controller then waits for the server to respond. If the destination responds with a confirmation that the data was successfully uploaded, then and only then are the rows that were uploaded deleted from the database. This may cause data to be uploaded multiple times if there is bad connectivity, but no data will ever get lost because of unscheduled interruptions. It is much more desirable to have data get repeated on the backend than have data go missing. And the repeated data points can easily be accounted for and properly handled by a client with greater resources, such as a PC or server. The upload controller is also implemented using Active Objects so that it can be run concurrently with the sensing so as not to prevent data from being collected.

The upload controller selects which access point to connect to based on an ordered list of predefined access points on the mobile phone in the order of last used. There is a high chance that the last access point that the phone used was a good one and that it will still work. But if not, the next one on the list has the next highest probability. However, if nothing can be connected to, or no known access points exist, Campaignr will pop up a dialog asking the user if they would like to choose one themselves. And if they would, another dialog will popup displaying the access points available on the phone and an option to scan for WiFi points if the hardware supports such. If a Symbian S60 phone is in offline mode, it will pop up a dialog asking the user if they want to allow anything that would emit radiation over a radio every time any connection wishes to be established, with no way of disabling the prompt. That makes trying to connect a phone with no SIM card to a WiFi access point problematic. Therefore Campaignr pops up a dialog at the first attempt to connect to the internet asking whether the user does want to continue connecting in Offline Mode. If not, Campaignr will not continue to connect until the user enables uploading themselves so as to reduce the number of annoying popups. The algorithm for handling connections is not the most optimal, but proper handling of connections and disconnection in a robust and automatic way is a research topic all its own.

If the user or author of the campaign knows that data are going to be collected in an area that will never have network coverage, they can specify that Campaingr either start with not trying to upload any data or write the XMLs that would've been sent over the internet to the phone memory so that they can then be later transfered to a PC and examined if that is all that is needed, or uploaded when a connection is finally established, since they are already in the proper format for uploading. This expands the working range of Campaingr greatly without much modification to Campaingr or special purpose code on the PC. There is a Python script that will extract the binaries from the XML files and base 64 decode them for easy viewing/listening. Since it is written in Python, it would even be possible to run the script on the phone itself in a pinch using the Python implementation for S60 phones.

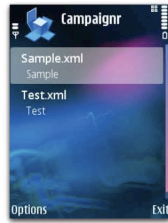

Fig. 5. The File List View displaying a couple of XML that Campaignr found.

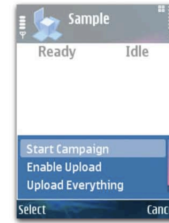

Fig. 6. The Main Menu showing options to start collecting data, start uploading data, and an option to quickly upload whatever is left in the database.

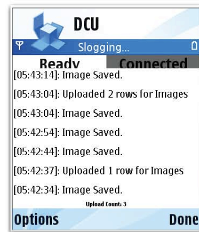

Fig. 7. The Main View after some data has been collected and uploaded.

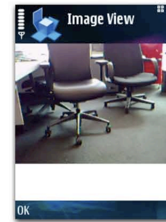

Fig. 8. The Image View that appears when the `viewfinder` tag is passed to the `image` sensor.

3.5 User Interface

The user interface is almost all handled by the main view, which talks directly to the main controller. The first screen that a user sees is a separate view from the main view that handles finding and displaying the campaign XML files in two specific directories, one on the internal memory and one on the external memory card. It is possible to have the same file in both places and they will not interfere with each other since the database that is created based off of the XML file is created in the same directory that the XML file lives. The user is able to scroll through the list of available campaigns and choose one to run. However if there is a file called `autoload` in either directory (with the external memory being accessed first) then that file is read and the campaign XML file it names is automatically loaded, bypassing the file list view completely. Normally, when the user is done with a particular campaign, they are taken back to the file list to be able to choose another one or exit Campaignr. But this behavior is changed to exit Campaignr from the main view if the campaign was selected by the `autoload` file. This is a very handy feature, along with the `startOnLoad` option in the XML, for when the phones are given to other people to use during a campaign. The user does not have to deal with remembering which campaign it is that they are supposed to run, as well as not having to push as many buttons to start collecting data.

After a campaign is parsed and loaded, the main view appears. It starts out with a blank screen where the name of the campaign is on the top as seen in Figure 4, unless the whole top part of the screen has been hidden by the `hide_status` tag specified in the XML file as seen in Figure 3. As data is collected or uploaded the status bar changes its text to provide status information (in the case of Figure 7: “Slogging...” (i.e. Uploading)). As actions occur, such as

taking a picture, recording audio, or successfully uploading data, they are prepended to the white text area in the main middle part of the screen with timestamps of when they happened so that the user can see whether things are still up and running and behaving properly. The `Options` and `Done` on the bottom of Figure 7 are labels for softkeys on the phone that are just below the labels. Pressing the right softkey will stop the campaign to allow the user to choose another, with the exception described above. The left softkey brings up a context sensitive list of options as seen in Figure 6. There is an option for either turning the data collection on or off, another option for turning on or off uploading. And another option appears only if the campaign is stopped. This option puts Campaignr into a mode that will upload all the data in the database through a series of upload cycles without collecting any new data. This feature allows the user to turn off uploading, if it has not already been disabled by an XML option, when the user knows there will be no connectivity, and then let the phone upload everything it has collected when the user has finally entered an area with connectivity.

The image sensor can bring up its own view when the `viewfinder` tag is specified for the `image sensor` as seen in Figure 8. The viewfinder allows the user to preview what the camera is seeing to frame their own shots and it treats the button thats used for initiating a manual data collection run as the shutter button since the phone is in all respects acting exactly like a digital camera. After the image is taken the camera view destroys itself and the main view is once again shown.

As discussed before and shown in Figure 3, the top status portion of the view can be hidden to give more room to the text area. The text area's font size can be increased for ease of reading. The `Options` label can be hidden, the left softkey disabled, and the right softkey functionality changed to completely exit Campaignr so that a user can not change anything if the campaign creator so desires when the `hide_options` tag is present in the XML file.

The microphone sensor has been given the ability to vibrate and/or beep when the recording starts and stops to experiment with providing extra and novel modalities of feedback to the user. The non visual cues help when the screen of the phone is not readily viewable. This is a nice feature that will eventually be implemented in the other sensors such as the image sensor. One common way that Campaignr has been used is to hang the phone around the neck of the user and let it automatically take pictures. If the phone were to vibrate every time it took a picture, it would remind the user that images are being taken so that they will be more conscious of their surroundings so they do not, for instance, accidentally go into the restroom while images are still being actively captured.

4 Related Work

In the past few years there has been a fair bit of previous work to provide a way of gathering data by mobile device[8–11] and lots of work utilizing mobile devices to gather data[3, 4, 12–19].

An application written in Symbian C++ that has been previously used for participatory sensing campaigns called Mobile Web Server, also known as Raccoon[20]. Raccoon is a port of the Apache web server to the Symbian S60 platform. It provides external access to the phone via standard protocols, allowing the phone to take and send images by visiting the mobile phone's url. This symplifies image collection. However the phone has to have connectivity, which limits the operating scope.

StarScape[21] is a J2ME based participatory sensing application that can be set up to collect data from a number of mobile phone sensors and automatically upload them to SensorBase. Because it is written in J2ME, it has the ability to be installed on a much wider set of

Table 1. Lifetimes while capturing and uploading an image every 30 seconds (as tested on the same Nokia N80 device).

StarScape	Raccoon	Campaignr
2h 31m	7h 44m	8h 53m

Table 2. Number of button presses required to start collecting data (counted from the application's start screen).

StarScape	Raccoon	Campaignr
Minimum		
1	3	0
Maximum		
10+	3	4

Table 3. Time to base 64 encode an image (as tested on the same Nokia N80 device)

Int. Mem.	Ext. Mem.	RAM
Python		
14s	152s	8s
Symbian C++		
.033s	.034s	0.011s

Table 4. Time to capture and save an image (as tested on the same Nokia N80 device).

StarScape	Raccoon	Campaignr
19.08s	5.73s	6.04s

mobile phones than applications written for Symbian OS[22]. However it has some significant limitations as described below.

5 Results

The creation of Campaignr was inspired by the use of Raccoon and StarScape as data collection applications. Raccoon was never intended to be used in this way but it is so well written that it could easily handle the heavy use. StarScape was rapidly developed to provide easy access to the native hardware sensors and helped to show what would and would not work with mobile devices.

These two predecessors were chosen to compare against Campaignr because they both have been used in situations typical to what Campaignr is designed for and they both also run on Symbian S60 3rd Edition mobile phones.

Campaignr performs on par with Raccoon and outperforms StarScape with regards to energy consumption and sensing speed as can be seen in Tables 1 and 4. One reason StarScape has such a lower lifetime than the other two applications is that the image capture functionality would keep getting into a bad state and not produce any more images which means that the application had to be restarted a few times during the experiment. That restarting required using the phone GUI and so the StarScape test had the backlight on for a much longer time than the other two. Being written in Java and having to run on a JVM also could account for some of the decreased lifetime.

Raccoon performs better than Campaignr because it never access the file system. Raccoon never stores any data locally, while Campaignr access the file system on every data collection. However the difference of 69 minutes in lifetime is not worth the trade off of lost samples when the connectivity is poor or no samples at all when it is nonexistent.

Another option for collecting data from mobile phones would be to write Python as discussed before. However the Python implementation for S60 phones was discovered to have a major defect when used in [1]. Trying to base 64 encode an image takes a significant amount of time; over two minutes in the worst case, eight seconds in the best case as seen in Table 3. The RAM time is implicitly included in the first two columns as the extra time is reading from

Table 5. Available sensors for each application. (All sensors for Raccoon, except for the camera, provided by Python Server Pages.)

StarScape	Raccoon	Symbian C++
	Hardware	
Camera GPS Microphone Video	Camera GPS Microphone	Camera GPS Microphone Video Bluetooth Stumbler Cell Tower ID Battery Level and Status Motion Band WiFi Stumbler
	Software	
Timestamp Text (username only)	Timestamp Text	Timestamp Text IMEI

the file system to memory. Encoding also has the side effect of locking up the entire phone during the process, even the screen is unable to refresh. That precludes sending binary data as an url encoded form over HTTP POST in Python. In Symbian C++ the same encoding takes only one hundredth of a second.

Campaignr can take from no button pushes to at most four to start collecting data as can be seen in Table 2. It takes one more button push is the worst case than Raccoon because Campaignr provides the ability to temporarily disable data collection without having to exit the application, and will then take less pushes to restart. Since StarScape always starts configured the same way it was when it quit, starting the collection process can be very quick, yet when a new campaign is run it takes a large amount of button pushes to configure.

StarScape can provide more feedback about what the application is doing, but the user interface is slow to respond to user input. Raccoon is very quick to respond to user input, but provides very little feedback. It is not possible to determine whether data is being successfully collected. Campaignr has both a low latency user interface and provides feedback into how it is performing.

Campaignr makes a few small trade offs in power and processing efficiency to provide robust data collection from a greater number of sensors than either StarScape or Raccoon provide. Campaignr has at least twice as many sensors as can be seen in Table 5[7].

6 Future

Campaignr is still under heavy development with new sensors and new functionality being added all the time. But at the same time Campaignr is under heavy use in many research projects, generating new feature requests and discovering un-before discovered bugs. Due to the anonymity requirements it cannot be exactly said who is using Campaingr since most of it is within the same research group.

[7] The Motion Band is a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer sensor that communicates via bluetooth

Table 6. Partial list of campaigns that have been run to date. (PEIR stands for Personal Environmental Impact Report. Monumento was an interactive art exhibition.)

Name	Sensors Used	Dates Run
Walkability	Camera, GPS, Timestamp, Cell ID, IMEI	2007.10.13
Rewind	Camera, Text, Microphone, Timestamp	2007.10.10
Lifeblogging	Camera, Bt Stumbler, Timestamp, Text, Cell ID	2007.09.15 - present
DietSense	Camera, IMEI, Battery Info	2007.08.06 - present
PEIR	GPS, Cell ID, IMEI, Timestamp	2007.07.20 - 2007.07.27
Monumento	Camera, GPS, IMEI, Text	2007.06.15
Sidewalk	Camera, GPS, IMEI, Cell ID, Timestamp, Text	2007.04.30 - 2007.05.06

Some of the upcoming features and sensors planned are: local processing of the data for such things as saving transmission costs, database and transmission encryption for privacy conscious campaigns, a WiFi stumbler sensor (which actually has already been implemented since the writing of this paper), and SMS integration.

To enable more flexibility, an upload controller that supports uploading in the JSON format will be written. The new format will allow the returned data to be less rigidly defined as well as make it easier provide extra metadata about what is being uploaded. It also opens up the possibility of uploading to more than just SensorBase.

Python would make an excellent wrapper around the work already done in Campaignr to provide more flexibility and expressiveness for campaign creation and execution.

Campaignr currently can only run on Symbian S60 3rd Edition devices which does limit the reachable audience. But there are no current plans to support other mobile device operating systems since the Symbian smartphone OS market share for Q2 2007 is 72% and 18.7 million Symbian smartphones were shipped by licensees in Q2 2007, up 52% from Q2 2006 (12.3 million)[26] There are millions of Symbian smartphones already deployed out in the world and their number is constantly growing. Right now porting Campaignr to other platforms is not worth the effort.

7 Conclusion

Having average everyday people collect information about their surroundings (participatory sensing) is becoming a hot topic. Campaingr provides a platform for easily collecting that data. It is flexible enough to handle many different kinds of data collection campaigns, supports a wide variety of sensors, and robust enough to be used by many groups while still in heavy active development. Campaignr was released on April 10, 2007 and was used for data collection for a separate research project within that first month and is still being widely used as can be seen in Table 6. Campaignr fills the need of wanting a way to collect data in a participatory sensing manner without having to learn how to program for the complex, and sometimes convoluted mobile embedded platform. Instead of creating throwaway applications that work for a single instance but break as soon as they need to be modified, Campaignr provides a way to collect data from different campaigns without having to deal with frustrating and time consuming development cycles.

Campaignr is an open source project[27] and more information about where to get, how to create campaigns for, and how to run Campaignr is available online[28].

References

1. Reddy, S., Schmid, T., Parker, A., Porway, J., Chen, G., Joki, A., Burke, J., Hansen, M., Estrin, D., Srivastava, M.: Urbancens: Sensing with the urban context in mind. In: UbiComp '06
2. Burke, J., Estrin, D., Hansen, M., Parker, A., Ramanathan, N., Reddy, S., Srivastava, M.B.: Participatory sensing. SenSys (2006)
3. Paulos, E., Goodman, E.: The familiar stranger: Anxiety, comfort, and play in public places. CHI (2004)
4. Reddy, S., Parker, A., Burke, J., Estrin, D., Hansen, H.: Image browsing, processing, and clustering for participatory sensing: Lessons from a dietsense prototype. EmNets (2007)
5. http://sensorbase.org
6. Symbian: How Do I Get My Symbian OS Application Signed? Version 2.5. (June 2007)
7. http://wiki.opensource.nokia.com/projects/PyS60
8. Froehlich, J., Chen, M.Y., Consolvo, S., Harrison, B., Landay, J.A.: My experience: A system for *In situ* tracing and capturing of user feedback on mobile phones. MobySys (2007)
9. Tuulos, V., Scheible, J., Nyholm, H.: Combining web, mobile phones and public displays in large-scale: Manhattan story mashup. Pervasive (2007)
10. Roduner, C., Langheinrich, M., Floerkemeier, C., Schwarzentrub, B.: Operating appliances with mobile phones - strengths and limits of a universal interaction device. Pervasive (2007)
11. Paulos, E., Joki, A., Vora, P., Burke, A.: Anyphone: Mobile applications for everyone. DUX (2007)
12. Sohn, T., Varshavsky, A., LaMarca, A., Chen, M.Y., Choudhury, T., Smith, I., Consolvo, S., Hightower, J., Griswold, W.G., de Lara, E.: Mobility detection using everyday gsm traces. UbiComp (2006)
13. Froehlich, J., Chen, M.Y., Smith, I.E., Potter, F.: Voting with your feet: An investigative study of the relationship between place visit behavior and preference. UbiComp (2006)
14. Hodges, S., Williams, L., Berry, E., Izadi, S., Srinivasan, J., Butler, A., Smyth, G., Kapur, N., Wood, K.: Sensecam: a retrospective memory aid. UbiComp (2006)
15. Matthews, T., Carter, S., Pai, C., Fong, J., Mankoff, J.: Scribe4me: Evaluating a mobile sound transcription tool for the deaf. UbiComp (2006)
16. Kindberg, T., Jones, T.: "merolyn the phone": A study of bluetooth naming practices. UbiComp (2007)
17. Bell, M., Hall, M., Chalmers, M., Gray, P., Brown, B.: Domino: Exploring mobile collaborative software adaptation. Pervasive (2006)
18. Burke, J., Estrin, D., Bell, G., Reddy, S., Ramanathan, N.: Sensing on everyday mobile phones in support of participatory research. SenSys (2007)
19. Srivastava, M., Hansen, M., Burke, J., Parker, A., Reddy, S., Saurabh, G., Allman, M., Paxson, V., Estrin, D.: Wireless urban sensing systems. Technical report, Center for Embedded Networked Sensyng, UCLA (April 2006)
20. Wikman, J., Dosa, F., Tarkiainen, M.: Personal website on a mobile phone. Technical report, Nokia Research Center (2006)
21. http://wiki.urban.cens.ucla.edu/index.php?title=StarScape
22. http://symbian.com
23. Nokia: S60 Plaftorm: Comparison of ANSI C++ and Symbian C++. Version 2.0. (May 2006)
24. Nokia: Symbian OS: Threads Programming. Version 1.0. (March 2005)
25. Nokia: Symbian OS: Active Objects And The Active Scheduler. Version 1.0. (August 2004)
26. Symbian: Symbian fast facts. Technical report, Symbian Ltd. (2007)
27. http://svn.urban.cens.ucla.edu/projects/campaignr/
28. http://campaignr.com