Research Article

MSF: An Efficient Mobile Phone Sensing Framework

Giuseppe Cardone, Andrea Cirri, Antonio Corradi, Luca Foschini, and Dario Maio

Dipartimento di Informatica, Scienza e Ingegneria (DISI), Università di Bologna, 40136 Bologna, Italy

Correspondence should be addressed to Luca Foschini; luca.foschini@unibo.it

Received 12 November 2012; Accepted 30 January 2013

Academic Editor: Nirvana Meratnia

Copyright © 2013 Giuseppe Cardone et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Recent evolutions in smartphones, today provided with several sensors, have the strong processing capabilities needed to extract from raw sensed data sensor meaningful high-level views of the physical context around the user. A new promising research area called mobile sensing promotes completely decentralized sensing based on smartphone capabilities only. However, current mobile sensing solutions are not very mature; yet, because they are based on ad hoc software solutions tailored to one specific technical problem (e.g., power management, resource locking, etc.), they are difficult to reuse and integrate in different projects, and they do not focus on the performance efficiency of the monitoring support. To overcome those limitations, this paper proposes Mobile Sensing Framework (MSF), a flexible platform to ease the development of mobile sensing applications through the definition of a common set of facilities that mask all low-level technical details in reading and processing raw sensor data. MSF has been optimized also to enhance performances for Android-based systems, and we report an extensive set of experimental results that assess our architecture and quantitatively compare it with a selection of other mobile sensing systems by showing that MSF outperforms them by presenting lower CPU usage and memory footprints.

1. Introduction

Current widespread off-the-shelf mobile platforms, such as Android and iOS, are broadening the traditional concept of mobile device to provide not only computing resource but also sensing capabilities, such as built-in sensors, including accelerometers, gyroscopes, GPSs, microphones and cameras. These new features make mobile devices powerful and complete sensing platforms to continuously watch and monitor the behavior of users who move and act in the physical world bringing with them their mobile devices. Moreover, it is possible to process on the mobile device large sets of locally collected raw data and to distill meaningful views of the activity currently done by the user, such as running, cycling, talking, and sitting, by exploiting signal processing and machine learning algorithms; in brief, we call the whole continuous sensing process as inferring user current activity. Many mobile applications can benefit these brand-new mobile sensing capacities and span different areas, from healthcare to homecare, from safety to smart grids and environmental monitoring, and many more.

After the initial hype, there are now several technical issues to be solved in designing mobile sensing applications and supports viable and valuable to the mobile market, mainly because there are still several open technical issues that affect the mobile sensing practice. First, most of the currently available solutions are considered vertical ones and make it difficult to reuse specific components, such as data gathering and energy management just to cite two typical horizontal facilities. Second, inferring user activity is a CPU-intensive task that requires retrieving raw data from sensors, preprocessing them to extract some synthetic characterizations of sampled signal periods (or features) and using these features to evaluate and infer actual activity [1]. In other words, mobile sensing is intrusive and risks to disrupt the overall user experience, especially because several mobile apps include multimedia services with strict soft real-time constraints. Third, monitoring tasks require intensive use of hardware sensors, computing resources, and storage to continuously gather, process, and save data; those activities can drastically reduce battery lifetime and should be carefully managed to control and minimize the mobile
sensing energy footprint. Fourth, although early projects used to off-load sensing and processing to external devices [2], modern sensing applications run both monitoring and processing directly on the smartphone, thus requiring a more careful management of concurrent access to both sensors and computing resources [3].

To address all previous issues, we propose the Mobile Sensing Framework (MSF) solution, a novel Android framework for mobile sensing that aims at offering app developers a set of attractive facilities and functions to quickly and easily design their own mobile sensing services. MSF exhibits several original characteristics. First, MSF is general purpose: its architecture includes some horizontal services, such as sensor control functions, raw data gathering, and power management to allow developers to focus on sensing logic and to easily wire their own data processing code into existing MSF skeletons, without having to deal with repetitive control and management tasks, demanded to the framework. Second, MSF is nonintrusive on user experience: it has been optimized for managing large streams of raw sensing data by carefully tuning and controlling concurrent access to system resources so to avoid any useless resource bind and to minimize additional processing overhead. Third, MSF is energy aware: it implements an efficient and flexible power management component that minimizes the impact of sensing applications on battery lifetime and includes several policies, both predefined and configurable by developers, to automatically control the duty-cycle of each sensing task. Finally, MSF is performant: to better underline this original aspect of MSF, the paper presents a thorough quantitative comparison with a selection of very close mobile sensing solutions to show that MSF outperforms other benchmarked systems in terms of performances and scalability.

This paper is organized as follows. Section 2 describes previous works useful to understand main issues in mobile sensing. Section 3 details the design guidelines stemmed from the analysis of previous work and the resulting architecture of MSF. Finally, Section 4 describes MSF implementation and assesses its performance by comparing it to an existing framework and to a barebone implementation. Section 5 concludes the paper and discusses future research directions.

2. Related Work

The mobile sensing trend has spurred many research projects that have focused on a variety of different problems, spanning from power-efficient signal processing to social sharing of sensed data. We believe that understanding the goals, the needs, and techniques of existing research efforts is the key to develop a lightweight mobile sensing framework that is useful and easy to use. Hence, in the following we report some of the major works in the mobile sensing literature presented, by starting with more simple single ones, such as applications based on a single sensor, and then moving on to multisensor applications, to conclude with more complex general-purpose sensing frameworks.

The first generation includes seminal works that leverage a single sensor, among the many available on the mobile device (accelerometer, microphone, light, etc.), to gather monitoring data and to use them to infer user current activity, such as walking, running, and standing. Usually, these are typically vertical “silos” applications that start from raw sensor data, go through a preprocessing stage, and end with a classification stage. A commonly used sensor is the accelerometer exploited to infer the current physical activity of the user [4, 5]; then, recognized activities can be used in multiple ways, to include promoting green behavior, monitoring fitness, and emergency detection. For instance, UbiFit measures the physical activity of users to nudge them towards using more environmentally sustainable means of transportation (e.g., walking versus driving) [2]. GymSkill is another example of a fitness mobile sensing application that monitors and assesses the quantity and quality of some physical activities that use standard fitness equipment [6]. About healthcare, PerFallID uses accelerometer data to detect user falls via a simple classification stage based on a threshold whose value is dynamically adjusted by using data collected from real users and occurred falls [7]. However, the accelerometer is not the only sensor used in many people-centric sensing applications; microphone is also a good source of information to make accurate inferences about people and environment. For example, the SoundSense realizes a high-level activity inference component that recognizes music, speech, and different ambient sound [8]. SpiroSmart, instead, is an iPhone healthcare app from an analysis user exhaling sound estimates breathing parameters, usually obtained via spirometer [9].

The second generation of mobile sensing applications explores the possibility to fuse data coming from multiple sensors toward different possible goals, such as either increasing classification accuracy or providing additional features. These proposals are still vertical systems but start to adopt more complex architectural solutions that may include horizontal services, such as in power management. Prominent examples of these multi-sensor applications are CenceMe, BALANCE, and BeWell [10–12]. CenceMe is a personal sensing system that allows users to share their activities with friends on social networks; it gets data from accelerometer, camera, and microphone and infers different socio/physical dimensions such as user activity (e.g., walking, biking, and running), disposition (e.g., happy and sad), and environmental conditions (e.g., noisy, hot, and bright) that can be automatically shared on popular social network such as Facebook and Twitter [10]. BALANCE, instead, based on input from accelerometer, barometer, GPS, light sensor, humidity, and microphone, aims at automatically estimating calories burst by users [11]. Finally, again addressing wellness, BeWell tracks three main wellbeing dimensions, namely, social interactions, physical activity, and sleep, estimated via inference over multiple sensors (mainly accelerometer and microphone) with the goal to give users easy-to-interpret feedbacks, evaluated against recommended values indicated by medical experts, about how their wellbeing changes over time [12].

All these research efforts, together with many ones we cannot cover here for space limitations, have generated enough momentum to push the development of the third generation of mobile sensing applications represented by
more comprehensive mobile sensing frameworks. A seminal work in this direction is the Funf Open Sensing Framework (in short Funf) [13]. Funf is an extensible sensing and data processing framework for Android providing a minimal set of reusable facilities for collecting, locally configuring, and uploading to remote servers a wide range of sensing activities. First of all, Funf divides sensors in hardware ones (e.g., accelerometer, microphone, GPS, etc.) and logical ones (e.g., recorded sound streams, application usage data, etc.) and defines data sources abstractions, namely, "probes", and Application Programming Interface (API) to gather raw sensing data. However, it presents several limitations; it provides only low-level sensing data access functions and does not support inferring higher-level activities; its horizontal facilities are very static, such as power management that only allows for the definition of configuration files indicating the query period of each probe; it does not include those micro-/macrooptimizations, such as object reuse, lightweight resource binding, that are extremely important to make a framework acceptable in terms of overhead, responsiveness, and resource consumption for the final user.

3. Design Guidelines and Logical Architecture

As in the previous section, designing a general-purpose mobile sensing is still a complex task that requires a deep knowledge about all common traits and issues of mobile sensing applications. From our knowledge and experience of this area, we have identified some main design guidelines for a logical architecture of our claimed fourth generation of mobile sensing applications: full-fledged sensing frameworks that include not only generalized sensing capabilities but also horizontal ancillary service, to ease the design of novel, even complex, activity recognition components, by efficiently taking care of all low-level system-/resource-management issues and by hiding unnecessary details.

3.1. Design Guidelines. First of all, mobile sensing applies to several application domains, each one with its own specific characteristics; sensing can be either continuous or sparse, classification can be either lightweight or complex, data can be processed either locally or not. Thus, the framework architecture should be modular and easy to use at two levels; application developers should be able to quickly create new applications based on raw data and/or already computed high-level inferences; library developers should be able to easily plug-in new components, such as support for new sensors and activity classifiers. To achieve both goals, it is crucial to adopt a layered architecture that clearly divides main framework levels: accessing sensors and system resources; inferring activities by processing sensed data; providing high-level abstractions to services to query and to register for specific activity recognition events.

Second, mobile devices must always be responsive to user input and should not cause any unexpected behavior, such as errors due to hardware resource locking. Therefore, the framework should be able to manage and control itself, namely, to adaptively tailor all sensing operations that might undermine and degrade user experience, by resolving all possible conflicts. For example, because the microphone resource can be acquired exclusively by one process sensing at a time, the framework should automatically switch-off all audio sampling sensing tasks whenever the user receives a call not to interfere with user expected phone-call behavior. Moreover, user transparency requires to hide the takeover to library developers without requiring a deep knowledge of low-level system issues.

Third, the framework should allow library and application developers to flexibly and dynamically change the framework behavior through easy-to-use configuration primitives and directives. Along that direction, the framework should provide a set of configurable management components, especially for sensor and power management. The sensor management should make it possible to dynamically reconfigure all ongoing activity recognition tasks, when applications/users decide activities on specific sensors. Similarly, since mobile devices have a limited battery capacity and continuous sensing can significantly reduce battery life time, the framework should support multiple energy saving strategies and adaptive duty-cycling approaches, and it should be possible to switch from one approach to another at runtime, without stopping ongoing sensing tasks.

Fourth, and finally, because mobile sensing applications rely heavily on CPU and memory hungry algorithms that can deteriorate smartphone performances, the sensing framework should be carefully designed and tailored to include any possible low-level optimization to reduce overhead and to limit the impact on local resource usage (CPU, memory, communications, bandwidth, etc.). In particular, used resources should be carefully bounded, and reuse of resources should be fostered, so as to avoid the execution of frequent and heavy garbage collection operations; hence, whenever possible, resources should be preferred reusable in pools and employed for both passive and active entities involved in the whole sensing process.

3.2. MSF Logical Architecture. The MSF goal is to provide a high level framework for the development of sensing applications that provide out-of-the-box classification algorithms capable of inferring high-level activities (e.g., walking, running, and cycling) from raw sensor data.

Let us all start by introducing some basic MSF concepts and abstractions. First of all, we name Input any source of sensing data (e.g., accelerometer, microphone, GPS), while a Pipeline is the component that encapsulates the application logic to gather, process, and meld together sensed data, collected from one or more Inputs, so as to evaluate—typically by exploiting specific classification algorithms and, possibly, also inference engines—high-level views of current user activity, namely, activity inferences, such as to recognize a specific user activity like sleeping, walking, and standing. Conceptually, Inputs and Pipelines are the basic building blocks to realize new MSF mobile sensing chains that continuously run and evaluate activity via inferences. According to our sensing framework self-control principle, there must be a way to interact with and, when necessary, to interrupt parts of
these sensing chains. The Management System is in charge of mediating MSF interactions with the external world, by handling two main types of events, namely, system events, triggered either by the system or by other apps (e.g., incoming call, battery running out, etc.), and user events, triggered by users (e.g., pausing all sensing tasks, disabling an input, etc.), and it controls and coordinates the interactions between Inputs and Pipelines.

To better ground these concepts and to illustrate the complexity of the management issues involved in realizing nonintrusive mobile sensing tasks, let us introduce a simple but real activity recognition scenario. Suppose that we want to develop a sensing application that uses a microphone and an accelerometer to recognize the following activities: walking, running, stationary silent, and stationary speaking user. In order to realize it, we need two Input objects to sense, respectively, the microphone and the accelerometer, and a Pipeline that encapsulates activity inference application logic. When the application is running, Input components read and deliver sensed data to the Pipeline, which returns the result of the activity recognition algorithm to the registered application. Now, let us assume that an application, such as the voice call application, needs to access the microphone (in general, an Input) to either take an incoming phone call or place an outgoing one. The need of an Input is typically signaled by Android applications through internal system events; the framework can register to receive those events and then can immediately release the microphone to allow users answering/making the call.

To be more precise, Figure 1 details all main interactions between framework components in the previous example. When the incoming call arrives, the audio-call app broadcasts an event to other interested apps; MSF, that handles all available system/user events (step 1 in Figure 1), dispatches it to our Management System that must stop sensing processes that require the microphone, release the microphone, and then reacquire it, and restart stopped processes when the call completes. In particular, the Management System triggers the following chain of actions: it sends a control message to the audio Input asking it to release the microphone resource (step 2); the Input pauses, releases the microphone, that can then be used for the phone call, and notifies this state change by broadcasting an internal Input event (within the MSF framework) that can be caught by all Pipelines using the microphone Input that can either pause themselves until the microphone is available again or keep running without that data (step 3); Pipelines that decide to stop notify their decision to the Management System that keeps track of the whole MSF internal state (step 4). When the call ends, the Management System wakes up the microphone Input and notifies all stopped Pipelines that the microphone is available again (step 5).

It is important to stress that our MSF Input management allows continuously accessing sensors and dynamically pausing them on external events, thus minimizing the impact on user-perceived responsiveness of their smartphone. In addition, let us remark that we use the same event-based management also to enable power management. In fact, to minimize power consumption of MSF means to pause and restart Inputs according to a given policy, from simple ones that periodically turn the sensors off to more advanced ones that, according to current executing conditions, modulate and tailor sensor activations. We are currently working on the power management component left out of the scope in this paper due to space limitations and because here we wanted to focus on the core sensing functionalities and computing performances of MSF framework.

After presenting the main MSF components, in the following, we detail the logical architecture and all entities of our framework that consist of two main subsystems: Sensing and Management of Figure 2.

The Sensing subsystem adopts a three-layered architecture and deals with all data gathering and data processing aspects; at the bottom layer, we find Inputs to gather sensing data and to wrap them in easy-to-manage local objects sent by a Bus to enable internal delivery of sensed data to all interested Pipelines; Pipelines are the core components of the middle layer to work on a global and efficient view; finally, at the top layer, the Dispatcher provides to interested apps the APIs to register to MSF to receive high-level inferences evaluated by Pipelines.

The Management subsystem, instead, coordinates and controls the whole interaction of the Sensing subsystem with the environment; the Interaction Layer is the listener that receives internal (i.e., framework) and external (i.e., system and user) events, while the Input Manager is the core management component that, according to Pipeline needs and currently monitored system/user situation, coordinates Input and Pipeline execution by propagating Input state change to Pipelines that can temporarily pause until the needed Inputs are available again. From this logical architecture, applicable to any mobile platform, we realized our MSF for the Android platform that is the most widely adopted one in mobile sensing, also because it allows sensor access even when the system is in standby, a key feature needed by continuous sensing systems and is not available on other mobile platforms (e.g., Apple iOS) [12, 14].

3.2.1. Sensing Components. This section describes some finer-level details of Sensing subsystem components. The Input component is the data source to enable sensing from local hardware and logical sensors; it gathers data and makes them available to Pipelines. All Inputs share the same interface; thus, MSF can instantiate them and manage their lifecycle by abstracting from their internal details. This allows third-party developers to easily develop and integrate new Inputs. Input instantiation is dynamically managed by Input Manager according to Input needs expressed by Pipelines through the Input Factory; then, Input Manager manages the whole Input lifecycle.

To facilitate programmers with a well-known pattern, the Input lifecycle mimics the one of Android components with seven states [15]. After the initial creation phase (see Created state in Figure 3), Input switches to Started state, in which it is configured and initialized before start reading sensor data; more precisely, this state makes the Input get all long-term resources to use for its whole lifetime span, such as
internal buffers to store sensed data. *Resumed state* is the actual execution state in which Input gathers sensing data and pushes them up to the Bus and eventually to Pipelines; then, when Input has to temporarily stop, it makes a transition from *Resumed* to *Paused*. Following the behavior suggested by Android, in *Paused state*, Input must release lightweight resources that can be easily reacquired afterwards, including used sensors, such as the microphone released during a phone call. Finally, in the *Stopped state* Input frees all allocated resources so that, in this state, Input can be either destroyed (to *Destroyed state*) or reactivated (to *Started state* again).

The second sensing component is the *Bus* (Figure 4). It realizes many-to-many distribution of sensed data samples from Inputs to all interested pipelines; to correctly dispatch
sensed data to Pipelines, it keeps track of both active Pipelines/Inputs and all dependencies between Pipelines and Inputs.

Focusing on the resource usage aspects, we reason on sample, the minimum chunk of sensed data returned by hardware/logical sensor, and encapsulate each sample in a DataBundle realized as a container object wrapping that tags the raw sensed data sample with additional details useful both for the Pipeline and for the internal resource management; typical information are Input type, sensing timestamp, and a reference counter. In particular, let us stress that Input sampling rate may cause a high rate creation of DataBundle objects, such as in common cases of using microphone and accelerometer with high sampling frequency; hence, Pipelines use them, and DataBundle objects could uselessly waste memory resources until they are collected by the garbage collector, thus wasting CPU cycles too.

Following our main design guidelines, MSF avoids that waste by using a DataBundle object pool to reduce the average memory footprint of the framework, by recycling already used DataBundle objects, and by using explicit reference counting. When the Input obtains a new sample from a sensor, it gets a free DataBundle from DataBundle pool and passes it to the Bus; then, the Bus initializes the reference counter to the number of Pipeline subscribed to that Input and delivers the DataBundle; afterwards, each Pipelines, once completed the sampling process, decrements the DataBundle counter; when the DataBundle has been used by all Pipelines, it is released back to the pool.

We have to better describe Pipelines, the general-purpose skeletons to be filled with specific activity inference application logic. Pipelines are designed to be self-contained and easy to dynamically instantiate, activate, and deactivate at runtime. Each Pipeline, identified via a unique global identifier, has to explicitly declare the Inputs it wants to subscribe to and has to follow the Pipeline lifecycle defined by MSF. The unique identifier, currently represented as the fully qualified package name, allows applications to unambiguously choose the Pipelines to use; about Input subscriptions, instead, each Pipeline has to statically declare them and then, at runtime, Bus and Input Manager uses those subscriptions, respectively, to deliver DataBundles and to signal the Pipeline of Input state changes. Finally, Pipeline lifecycle is similar to Input one. Pipelines are first created; then, while started and resumed they can receive raw data from Inputs and can output inferences; Pipelines can also pause, by temporarily releasing resources, and can stop for longer periods; if they are not needed anymore, they are destroyed. At the same time, it is important to note that while Input lifecycle is mainly driven by external user and system events via the Input Manager, Pipeline lifecycle is self-contained in the private Pipeline code that controls and decides needed strategies for state transitions autonomously, to allow different policy coexistence. For instance, a Pipeline that takes data from two different Inputs may be able to work even if one of them is paused or stopped by generating approximated inferences; another Pipeline may not work without the data from all Inputs and, hence, should switch to paused or stopped state as soon as any of its Inputs stops.

The very loose constraints imposed by MSF allow the development of arbitrarily complex Pipelines; however, it is up to developers not to abuse by designing Pipelines CPU and memory hungry that, in their turn, could cause excessive battery usage and worsen final user experience.
The last Sensing component is the Dispatcher that realizes a many-to-many distribution model to deliver activity inferences from Pipelines to interested apps. New apps present to the Dispatcher their interest in receiving activity inferences from a specific Pipeline, and the Dispatcher calls them back, as soon as a new relevant inference is available. Thanks to MSF dynamic management of Pipelines and Input, apps can also register for Pipelines that have not been instantiated because there are not interested clients yet; in that case, Dispatcher boots a new sensing chain by creating a new Pipeline; symmetrically, if all apps deregistered from a Pipeline, the Dispatcher frees systems resources by shutting it down. Pipeline creation and destruction triggers the creation/destruction of Inputs, managed by Input Manager.

3.2.2. Management Components. The Input Manager is the key Management subsystem component that manages Input lifecycle from instantiation to shutdown thus, indirectly, influencing also Pipelines lifecycles. When the Dispatcher creates a new Pipeline, the Input Manager starts all needed Inputs; correspondingly, when a Dispatcher is destroyed, the Input Manager also frees all Inputs that are no longer used. In addition to Inputs de-/allocation, the Input Manager is also responsible for driving the whole Input lifecycle by triggering their pause, resume, stop, and restart operations, based on event notification received from the Interaction Layer.

The Interaction Layer receives system- and user-events and reports them to MSF so as to avoid their interference with the expected behavior of the mobile phone, in the sense of nonintrusiveness. The Interaction layer also supports other events, such as low-level battery warning, screen going on or off, and other user-defined events (that allow users to stop sensing whenever they want). The Interaction Layer receives all these events and passes them to the Input Manager, which then accordingly manages the lifecycle of inputs. The Interaction Layer acts on a strict event-action basis and its support for arbitrary events provides the building block to develop new power management policies that drive the duty cycling of Inputs. The current MSF power management policy leverages the Interaction Layer to implement a simple duty-cycle policy that periodically pauses and, after a while, resumes all active sensors; however, the same architecture allows to easily integrate more complex policies that selectively pause Inputs by adapting their decision to the current usage context [16–19].

4. Implementation and Experimental Results

This section presents MSF implementation and shows an interesting selection of experimental results that compare MSF with the Funf framework that is the closest one available in the state-of-the-art literature [13].

MSF has been realized as a self-contained app that runs on Android platform and is compatible with version 2.2 up to version 4.1. The current MSF implementation includes Input objects for the following sensors: accelerometer, microphone, magnetometer, gyroscope, and light sensor; in addition, it provides two Pipelines, one based on accelerometer data and one based on audio data. The accelerometer Pipeline identifies the current physical activity of the user, namely, resting, walking, and cycling, by running a classification algorithm that analyzes some signal features: maximum, minimum, average, standard deviation, and root mean square over the three accelerometer axes. The audio Pipeline recognizes human voice based on some time-domain and frequency-domain features typically considered in the related literature, namely, L1-norm, L2-norm, L-inf norm, Fast Fourier Transform, power spectral density across five different band ranges, and Mel-frequency cepstral coefficients [10, 12, 14]. These pipelines are representative of real-world workloads, because similar functionalities have been used by existing works based on continuous mobile sensing.

We compared MSF performances for these two Pipelines, with other two implementations of the same Pipelines: a native Android and a Funf solution. Native solution does not rely on any external library and runs the barebone minimum code to perform sensor sampling and the feature extraction of audio; this solution represents the reference to evaluate memory and CPU overhead of other approaches, namely, MSF and Funf. Funf-based solution, instead, uses two probes (data sources), namely, AccelerometerSensorProbe and AudioFeaturesProbe, that implement the same MSF sensing chain for the two considered Pipelines. In order to make the test fair and comparable, we disabled the Funf default feature that dumps all data to a local database because it slows down and worsens system performances, especially for high sensing frequencies. Let us also stress that the audio feature extraction code that is the most CPU intensive task has been coded exactly in the same way for all the compared solutions.

We tested the three implementations (native, Funf, and MSF) using different sensor sampling frequencies. Audio was tested at 8 kHz and 44 kHz sampling frequencies, while accelerometer was tested using the three sampling frequencies made available by standard Android APIs, from lowest to highest frequency: SENSOR_DELAY_NORMAL, SENSOR_DELAY_GAME, and SENSOR_DELAY_FASTEST. All test have been run on a Samsung I9100 Android device featuring a dual core ARM Cortex-A9 processor running at 1.2 GHz and 1 GB RAM [20].

Our first set of experiments measures the introduced overhead in terms of CPU. Figure 5 shows obtained average CPU usage for each test setting, for different sampling frequencies and with different active Pipelines (only audio, only accelerometer, or both); each experiment has been repeated 33 times, and black vertical error bars report 95% confidence intervals. In general, the results show that MSF has a very little overhead compared to native, barebone implementation, thanks to its careful management of resources. Focusing on the second test that processes audio at 44 kHz, the MSF CPU usage is even smaller than that of the native solution one. We believe that is because the thrifty resource management of MSF recycles many of the internal objects and especially the byte arrays that store audio samples wrapped in DataBundles; however, the simple solution create a lot of new objects for each sensing cycle that have to be routinely freed by the Garbage Collector, thus causing increased CPU usage. That
effect is more noticeable when audio is sampled at 44 kHz because that high frequency stresses more the sensing code.

The third, fourth, and fifth tests report CPU load as the accelerometer sampling data rate raises from 5 samples/sec (SENSOR_DELAY_NORMAL), to 50 samples/sec (SENSOR_DELAY_GAME), and finally to 100 samples/sec (SENSOR_DELAY_FASTEST). Let us remark that the Pipeline that extracts features from the accelerometer signal needs to accumulate a certain amount of data before running; when it runs, it causes a surge in CPU load; however, it takes more time to collect enough data at slow data rate setting, and thus the average CPU load decreases as the accelerometer data rate gets slower. For this reason, when the accelerometer is set at 5 samples/sec (third test), it takes a relatively long time for the Pipeline to trigger its feature extraction algorithm, thus causing a low CPU load, that is about 0.2% for both the barebone and MSF implementations. At the same rate, the higher load of Funf is caused by DataBundles usage.

Finally, the last and sixth tests realistically stress barebone, MSF, and Funf sensing capabilities when sampling both the accelerometer and the microphone at the same time. Collected results confirm the low overhead of MSF; the barebone native solution causes 9.9% CPU load on average, while MSF takes 11.8%, only 1.9% more. On the other hand, Funf CPU load is 18.9%, almost the double compared to the cheapest solution; that big difference mainly is due to internal Funf architecture that does not pool objects by completely relying on Android Bundle objects that are easy to use but introduce very high overhead [21].

Our second set of experimental results assesses, under the same experimental conditions, memory usage for MSF, Funf, and native Android; in particular, we have used the heap dump feature of the Dalvik Debug Monitor Server (DDMS) provided by the Android SDK that takes a snapshot of current heap status for a running application by describing the live set of objects allocated at the moment the snapshot was triggered.

Figure 6 shows the heap size in each test settings and highlights the very good performance of MSF. Funf and native Android solution have a very similar approach to memory management based on creating new objects for each sensor sampling and letting the garbage collector remove them, and thus they have very similar heap usages; MSF optimized object pooling, instead, significantly reduces the average heap size by up to 2 MB compared to them. This good improvement over other solutions is even more important considered the strict heap size constraints that Android enforces by default. On lower-end smartphones, the maximum allowed heap size is as low as 16 MB; hence, MSF frees valuable memory resources that can be more fruitfully exploited to increase activity inference tasks.

We also compared MSF heap footprint with the one of an empty Android application (not shown in Figure 6) that, on our test device, was 8.1 MB; as Figure 6 shows, except when dealing with high quality audio, MSF footprint is always very close to that value thus confirming the limited overhead introduced by our framework.

5. Conclusion and Future Work

This paper presented MSF, a general-purpose high-performance framework for mobile sensing. The modular design of MSF presents two main architectural advantages over existing solutions; it makes easy for application-level
developers to exploit the intelligent MSF inferences and allows signal processing and machine-learning experts to quickly add new Pipelines neglecting all additional technical complexities and details of mobile sensing. In addition, as demonstrated by shown experimental performances, MSF design choices allow a very efficient resource management that makes MSF a powerful, easy-to-use, and flexible mobile sensing platform.

Based on these results and after collecting feedbacks from early adopters, we are already working to further expand MSF capabilities by focusing on several directions. First of all, we are realizing additional state-of-the-art classifiers to provide reliable inferences about current activities; at the same time, we are also working on providing additional support utilities, such as automatic data upload and data analysis tools. Finally, the current MSF implementation takes care of multiple sensing Inputs and Pipelines, but it does not support sharing of collected data among different applications that may have different requirements on sensed data quality, such as acquiring the same information at different rates, with different accuracies; to support multiple applications with differentiated sensed data quality requirements, we are adding a new component able to collect all sensing requests and to configure Inputs and Pipelines by reconciling their quality levels.

References

[1] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, “The Jigsaw continuous sensing engine for mobile phone applications,” in Proceedings of the 8th ACM International Conference on Embedded Networked Sensor Systems (SenSys’08), pp. 71–84, Zürich, Switzerland, November 2010.

[2] S. Consolvo, D. W. McDonald, T. Toscos et al., “Activity sensing in the wild: a field trial of UbiFit Garden,” in Proceedings of the 26th Annual CHI Conference on Human Factors in Computing Systems (CHI ’08), pp. 1797–1806, Florence, Italy, April 2008.

[3] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, “A survey of mobile phone sensing,” IEEE Communications Magazine, vol. 48, no. 9, pp. 140–150, 2010.

[4] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, “Activity recognition using cell phone accelerometers,” SIGKDD Exploration Newsletter, vol. 12, pp. 74–82, 2011.

[5] T. Brezmes, J. L. Gorricho, and J. Cotrina, “Activity recognition from accelerometer data on a mobile phone,” in Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part II: Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living, pp. 796–799, Salamanca, Spain, 2009.

[6] A. Möller, L. Roalter, S. Diewald et al., “Gymskill: a personal trainer for physical exercises,” in Proceedings of the 10th IEEE International Conference on Pervasive Computing and Communications (PerCom ’12), pp. 213–220, Lugano, Switzerland, 2012.

[7] J. Dai, X. Bai, Z. Yang, Z. Shen, and D. Xuan, “PerFallD: a pervasive fall detection system using mobile phones,” in Proceedings of the 8th IEEE PerCom Workshop on Pervasive Healthcare (PerHealth ’10), pp. 292–297, Mannheim, Germany, April 2010.

[8] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, “SoundSense: scalable sound sensing for people-centric applications on mobile phones,” in Proceedings of the 7th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys ’09), pp. 165–178, Kraków, Poland, June 2009.

[9] E. C. Larson, M. Goel, G. Boriello, S. Heltshe, M. Rosenfeld, and S. N. Patel, “Spirosmart: using a microphone to measure lung function on a mobile phone,” in Proceedings of the ACM Conference on Ubiquitous Computing (UbiComp ’12), pp. 280–289, Pittsburgh, Pa, USA, September 2012.

[10] E. Miluzzo, N. D. Lane, K. Fodor et al., “Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application,” in Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys ’08), pp. 337–350, Raleigh, NC, USA, November 2008.

[11] T. Denning, A. Andrew, R. Chaudhri et al., “BALANCE: towards a usable pervasive wellness application with accurate activity inference,” in Proceedings of the 10th Workshop on Mobile Computing Systems and Applications (HotMobile’09), pp. 1–6, Santa Cruz, Calif, USA, February 2009.

[12] M. Lin, N. Lane, M. Mohammad et al., “Bewell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization,” in Proceeding of the Wireless Health Academic/Industry Conference (Wireless Health ’12), San Diego, Calif, USA, 2012.

[13] N. Aharony and W. Gardner, 2012, Funf Developer Site, http://www.funf.org.

[14] N. Ramanathan, F. Alquaddoomi, H. Falaki et al., “Ohmage: an open mobile system for activity and experience sampling,” in Proceedings of the 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth ’12), San Diego, Calif, USA, April 2012.

[15] Google corp, 2012, Android Developers Site, http://developer.android.com/index.html.

[16] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow, “SociableSense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing,” in Proceedings of the 17th Annual International Conference on Mobile Computing and Networking (MobiCom ’11), pp. 73–84, Las Vegas, Nev, USA, August 2011.

[17] P. Bellavista, G. Cardone, A. Corradi, and L. Foschini, “The future internet convergence of IMS and ubiquitous smart environments: an IMS-based solution for energy efficiency,” Journal of Network and Computer Applications, vol. 35, no. 4, pp. 1203–2012, 2009.

[18] P. Bellavista, A. Corradi, M. Fanelli, and L. Foschini, “A survey of context data distribution for mobile ubiquitous systems,” ACM Computer Surveys, vol. 44, pp. 1–4, 2012.

[19] G. Cardone, A. Corradi, L. Foschini, and R. Montanari, “Socio-technical awareness to support recommendation and efficient delivery of IMS-enabled mobile services,” IEEE Communications Magazine, vol. 50, pp. 82–90, 2012.

[20] Samsung corp, 2012, Samsung I9000 Technical Specifications, http://www.samsung.com/uk/support/model/GT-I9000HKD-XEU-techspecs.

[21] C. K. Hsieh, H. Falaki, N. Ramathan, H. Tangmunarunkit, and D. Estrin, “Performance evaluation of android IPC for continuous sensing applications,” in Proceedings of the 12th workshop on Mobile Computing Systems and Applications (HotMobile ’12), San Diego, Calif, USA, February 2012.
