Sensor Fusion of Physical and Social Data Using Web SocialSense on Smartphone Mobile Browsers

Thomas Phan, Swaroop Kalasapur, and Anugeetha Kunjithapatham
Samsung Research America – Silicon Valley
San Jose, CA USA
{thomas.phan, s.kalasapur, anugeethak}@samsung.com

***Abstract*— Modern smartphones offer a rich selection of on-board sensors, where sensor access is typically performed through API calls provided by the phone's operating system. In this paper we evaluate the viability of implementing sensor processing entirely in the Web browser layer with Web SocialSense, a JavaScript framework for Tizen smartphones that uses a graph topology-based paradigm. This framework enables programmers to write personalized, context-aware applications that can dynamically fuse time-series signals from physical sensors (such as the accelerometer and geolocation services) and social software sensors (such as social network services and personal information management applications). To demonstrate the framework we implemented components for physical sensing and social software sensing to drive two context-aware applications, ActVertisements and Social Map.**

***Index Terms*—smartphones, sensors, activity recognition, social networking, sensor fusion, mobile web, context-awareness**

I. Introduction

Modern smartphones, including those running Android, iOS, and other mobile operating systems, provide a variety of programmatically-accessible on-device sensors, such as an accelerometer and gyroscope, as well as location-positioning available via GPS or network-signal trilateration. Thanks to this rich selection of sensing hardware and APIs to access them, much research work has focused on context-aware smartphone applications that can acquire and process sensor signals in order to produce inferred conclusions about the user's environmental state. Work in this area includes activity recognition [15], fitness tracking [9], and data gathering [5].

Enabling contextual awareness with sensor inferencing requires accessing the smartphone sensors through native library calls specific to the phone's operating system. Additional complexity may arise due to the need for developers to write in Objective-C on iOS and in Java or C/C++ on Android to invoke the sensors. Because both the sensor libraries and the programming languages differ in important ways, writing cross-platform software is difficult, with this difficulty becoming an increasingly larger obstacle as more sensors and APIs become available.

In this paper we evaluate the viability of implementing sensor acquisition and processing at a higher, cross-platform layer: the Web browser with application code written using JavaScript and HTML5, an approach apt for modern mobile operating systems like the Linux Foundation's Tizen [21] and Mozilla's Firefox OS [7]. These operating systems provide a platform supporting rich Web applications as first-class citizens with sensor accessibility exposed by corresponding APIs. While the feasibility of browser-based applications has already been demonstrated on desktops, we explore the novel notion of implementing sensor-driven applications entirely at this layer on a smartphone.

We developed Web SocialSense, a framework for developers to write compelling, context-aware applications for Web browsers running on smartphones. The framework itself is written entirely in JavaScript and provides the underlying components for applications to take advantage of multiple sensors. *Physical sensors*, such as the accelerometer and geolocation services, are well known, while what we call *social software sensors* are not. These latter sensors can be written to query online services, such as those from social networks like Facebook. For both physical and social software sensors, the data being acquired exhibits time-series behavior, and when coupled together, they can form the basis for useful and interesting end-user applications. To process such multi-source data, Web SocialSense presents a graph topology-based programming paradigm to the programmer: *source*, *processor*, and *fusion* nodes are connected by edges implemented by delay-tolerant JavaScript event buffering. To demonstrate the feasibility of our framework, we implemented two context-aware Tizen applications: *Social Map* shows contextual maps and routes, and *ActVertisements* shows activity-related ads.

This work makes the following contributions:

- We identify the challenge of running sophisticated sensor-based applications in mobile Web environments.
- We show that a fusion of physical sensors and social software sensors enables a unique opportunity to write personalized, context-aware user applications.
- We provide a reference implementation of our Web SocialSense framework and two demonstration applications, evaluating their performance to show the viability of sensor processing at the JavaScript layer.

The rest of this paper is organized as follows. In Section 2, we describe related work. In Section 3, we present the design of Web SocialSense, and in Section 4, we discuss experimental results. We conclude the paper in Section 5.

II. Related Work

To the best of our knowledge, Web SocialSense is the first Web-based framework that enables developers to fuse dynamic data from physical sensors and social software sensors in near-real-time. The following are some high-level research areas that are relevant to our work.

Sensor data abstraction: Previous researchers have looked into activity recognition, the process of abstracting raw physical sensor data to derive the user's activity, gesture, or locomotion. Early work in this field used custom hardware equipped with accelerometers (e.g. [3], [13]), while more recent work has used smartphones (e.g. [2], [9], [12], [15], [22]). Our work differs in that we fuse inferred activities with social data and implement the system entirely in the Web browser layer. Researchers have also looked at interpreting smartphone sensor readings to suggest automatic social network updates [17]; our approach differs in that we use social updates as inputs rather than produce them as outputs. The work in [4] also uses social inputs but is performed on a server for the purpose of accumulating provenance data.

Social data analytics: Many companies analyze real-time social feeds (e.g. Facebook and Twitter) with varied goals such as performing sentiment analysis to determine brand popularity and advertising impact. However, they do not incorporate any physical sensing other than geolocation to better understand the user's behavior and interests.

Component-based framework: We use a graph paradigm to modularize components and direct data flow from upstream sources to downstream sinks. Graphs are naturally suited for this type of problem and have been used in many contexts, such as distributed computing [10], AI [23], and audio processing [20]. Other component-based frameworks such as OSGi, COM, and EJB also offer modularization but are more heavy-weight.

III. Web SocialSense Design

A. Overview

While mobile sensing applications have proliferated on native OS platforms like Android and iOS, an open question remains regarding the viability of sensor-driven context-aware applications running at the Web layer, a cross-platform target that obviates the need for developers to code against a fixed native mobile platform. With the availability of modern smartphone operating systems supporting Web applications executing as first-class programs, we looked to explore this research and engineering challenge.

We created Web SocialSense, an event-driven, single-threaded, object-oriented JavaScript library that provides a framework for software developers to write compelling sensor-processing applications using a graph topology-based model. Application development using Web SocialSense entails acquiring digital signals from sensors, processing the data, generating JavaScript events that call back into top-level application logic, and returning results to the user interface.

As shown in Fig. 1, the Web SocialSense framework and user applications are written entirely in JavaScript and are run inside a Web Runtime (WRT) environment such as the one in Tizen, an approach that inherits many legacy advantages and disadvantages. As with any browser-based application, the "write once, run anywhere" goal allows developers to write writing code in cross-platform JavaScript, thereby hiding the heterogeneity of the operating system and hardware. For example, Google Maps and Docs are executed entirely within a Web browser on top of different operating systems. Likewise, Web SocialSense uses a WRT as an abstraction layer to hide underlying sensor acquisition heterogeneity. The cost of this approach is potentially less efficient performance compared against native software, but this factor is being actively addressed through advances in modern just-in-time JavaScript compilation techniques [19].

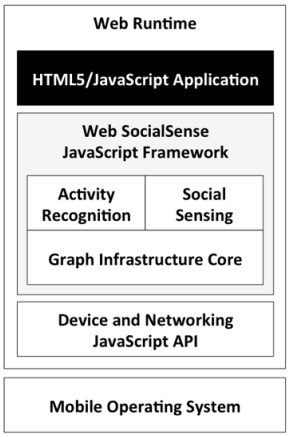

Fig. 1. Web SocialSense high-level components on Tizen.

B. Graph topology-based framework

Our main goal in developing Web SocialSense was to reduce the level of complexity and software development effort needed to create novel context-aware applications for WRT systems. Contextual awareness of the user's physical and social state can be achieved through various types of sensing, and we found that the flow of sensor acquisition and processing lends itself well to graph structures that specify how data moves from sensor sources to the final user-facing application. As a result, we designed Web SocialSense to follow a modular graph topology-based paradigm.

This flexible architecture provides abstractions over its components at various granularities, thereby allowing sub-graphs to be developed independently and then assembled together to realize new applications. For example, developers with deep knowledge of sensor data acquisition and statistical signal processing can build individual source nodes, while others with expertise in sensor fusion and data mining can configure the nodes together into a graph and return inferred results in the form of events. By maintaining components in this manner, the framework achieves both modular design and abstraction over implementation details.

Additionally, the framework offers introspection facilities for application developers to programmatically determine the events supported by the graphs. Developers can also control the lifecycle of the graph by starting and stopping the individual nodes as well as the entire graph.

1) Web SocialSense Architecture

The framework supports the creation of directed acyclic graphs; one such configuration is shown in Fig. 2. Each graph exposes one or more output events that are generated as a result of the processing paths within it; for example, our activity recognition component produces events with

meaningful predicate names such as "IsWalking" and "IsDriving." Application developers can then write code to listen for these events and act upon them.

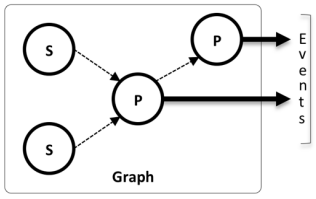

Fig. 2. Example graph configuration, where S indicates a source node and P indicates a processor node.

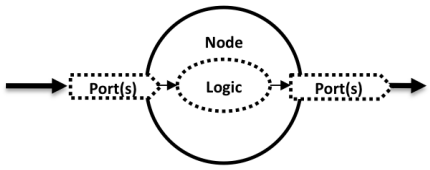

Fig. 3. Architecture of a node.

A developer builds a graph by defining the directed relationship among a set of nodes. A node is a logical entity responsible for performing a specific task in the framework, where each node can have zero or more incoming and one or more outgoing edges representing the corresponding flow of data. Fig. 3 shows the architecture of an individual node. A node that only has outgoing edges is a *source* node. Typical examples of such nodes are those responsible for capturing input from sensors such as the accelerometer, gyroscope, and geolocation services. A node that has both input and output edges is referred to as a *processor* node, which is generally implemented to read incoming data and perform processing to produce results that can then be consumed by other downstream nodes or be exposed directly to the top-level application, which acts as the final sink. Examples of processor nodes are those responsible for actions such as data filtering, fusion, and running specific algorithms such as FFT.

Two nodes can be connected together by a directed edge through which an upstream producer sends data to a downstream consumer. Edges are attached to nodes through a data structure that we call a port, where ports on all outgoing edges are implemented as a JavaScript typed array buffer in which the produced results can be enqueued for consumption by their down-stream counterparts.

2) Working with the Web SocialSense Framework

The first step in using the framework is to identify the necessary component functionality for a particular application. After needed requirements are identified, a developer can include existing subgraph libraries (such as our nodes that read and process physical sensors or social sources) or construct entirely new nodes.

Once all the necessary component nodes are ready, a graph can be constructed by connecting the nodes with directed edges, where each edge carries events with user-friendly names that, along with API documentation, can help express the semantics of the nodes. The developer can then package the resulting graph as a subgraph library that others can reuse.

C. Activity recognition with physical sensing

One library that we provide is a subgraph for reading physical sensors and processing the data to derive the user's physical context. As mentioned in Section 2, previous research work in activity recognition has concentrated on building machine learning training and classification systems where the classification is evaluated either offline or in a native smartphone application that can take advantage of OS API calls and faster execution. Our system operates similarly by reading sensor input and producing a classified output activity. However, we note that our focus is not core research in activity recognition; rather, we look at activity recognition as a packaged component that can be implemented in a WRT using our graph-based framework. The output of this library can then be incorporated by other downstream applications.

In an initial **training phase**, we built a machine learning model that maps sensor input to a classification label representing human physical activity. To collect data, we had users perform various activities while an Android application recorded the accelerometer at 32 Hz to get 3-dimensional acceleration vectors with all data converted to units of *g*, gravitational acceleration. The app also provided a user interface for users to self-label their activity to be one of running, walking, driving, and idling, where these specific activities were chosen to support the activity requirements of our top-level Social Map and ActVertisement applications. The application also allowed annotatation of the phone's on-body placement. We eliminated phone orientation differences by computing the acceleration vector's Cartesian length.

Once data was collected, we extracted features and built a classification model offline. Features were taken from a 256-sample sliding window with 50% overlap. We extracted the following features from each window: the frequency with the highest magnitude, the highest magnitude value; the weighted frequency mean (where each frequency bin is weighted by its magnitude) of the top-5 highest magnitude frequencies; the weighted frequency variance of the top-5 highest magnitude frequencies; and the on-body location of the phone.

Fig. 4 shows a screenshot of our diagnostics tool to evaluate the digital signal processing and classification. One window of 256 samples is shown in the second-from-the-top panel. We further applied a Hamming window function to smooth the tail ends, as shown in the third panel of Fig. 4, thereby reducing high-frequency components that result from abruptly truncated ends. For each window we computed the magnitude frequency response using an FFT, where the sample size of 256 provided sufficient granularity of 0.125 Hz per frequency bin in the resulting frequency spectrum. One magnitude response spectrum is shown in the fourth panel.

From this data we built a machine learning model that classifies each window. We used the WEKA data mining software [8] to build a C4.5 decision tree that we serialized to JSON format so that it could be used in the classification phase in the browser. One resulting classification is shown in

the top panel of Fig. 4, which indicates that the samples for the shown window correspond to a running activity.

We collected 356,096 training samples from five participants, resulting in 1391 windows. 10-fold cross-validation testing showed *an accuracy of over 95%*, which is competitive with previous work that classified a similar set of labels using a smartphone (e.g. [12] reported over 90%).

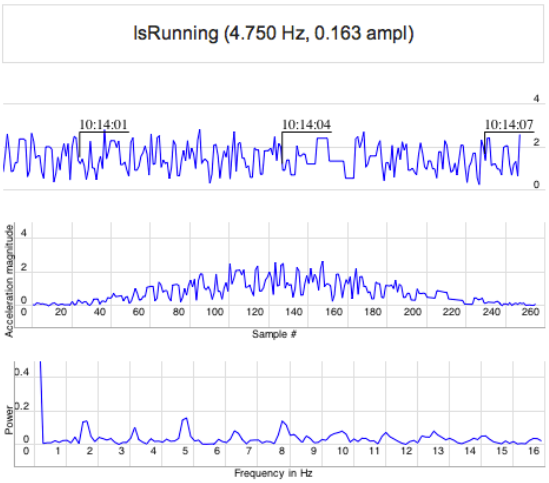

Fig. 4. A screenshot of our activity recognition and signal processing diagnostics tool written as a Web application with Web SocialSense. The top panel shows the resulting IsRunning activity predicate after performing classification of the raw accelerometer time-domain sample window, which is shown in the second panel. The third panel shows the samples after applying a Hamming window. The fourth panel shows the magnitude frequency response. The classification model uses some features from the frequency domain.

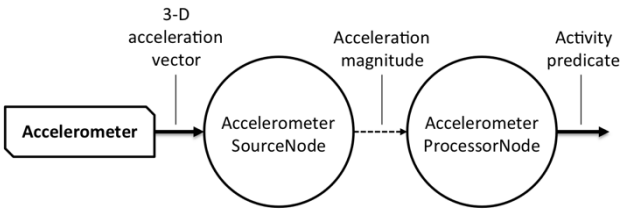

Fig. 5. Subgraph library for performing physical activity recognition.

The **classification phase**, performed online and in realtime on the smartphone, comprises two Web SocialSense nodes, as shown in Fig. 5. The first node, AccelerometerSourceNode, is responsible for retrieving accelerometer raw data using the same 32 Hz sampling rate used in the training phase. Reading the accelerometer at the WRT layer is problematic because sensor APIs are in the process of converging, but AccelerometerSourceNode hides underlying heterogeneity by internally checking and using the correct API. Our implementation supports the motion APIs from W3C (which is supported in Tizen) and Mozilla.

The online classification of accelerometer data to activity labels is performed by AccelerometerProcessorNode, which runs the same series of steps that was discussed for the offline training phase. The output JavaScript event is an user-readable activity predicate (e.g. "IsWalking" and "IsDriving") that downstream applications can use.

D. Social software sensing

We also provide a second subgraph library that performs social software sensing, a term we use to refer to acquiring data from sources related to the user's social connections, groups, schedule, and communication patterns. Specific social sensors can access:

- Social Networking services (SNS) such as Facebook, Twitter, Foursquare, and LinkedIn.
- Personal Information Management applications on the user's smart phone such as the address book, calendar, to-do lists, and reminders.
- Communication logs on the user's smartphone, such as the call history, instant messaging, and e-mail logs.

Clearly, user privacy is an important concern, and thus no action is taken unless the user specifically opts-in to this type of sensing. A fully commercialized product would further provide opting-in at finer granularities.

Social sensors observe activities, such as check-ins, status updates, comments, photo uploads, and "likes," and then reads both textual content (such as the body of a comment) as well as metadata (such as the time-stamp). This data can be created by either the user or his contacts and may be structured, semi-structured, or unstructured in form. Furthermore, data can be dynamic with frequent updates or relatively static with rare updates. Due to the varied forms of data that can be retrieved from these social sensors, different types of processing may need to be applied to extract meaningful information or hints about the user.

With respect to our graph framework, these social sensors are the upstream sources and components that use the data are downstream processors. Our Facebook social sensor provides parameters that allow the developer to specify login credentials and a polling rate, with rates anywhere between 30 seconds to several minutes being typical. On a mobile smartphone deployment, developers should implement their polling with careful consideration for battery usage [1]. An alternative approach would be the use of push notifications.

In our work we use only structured data returned via SNS APIs and operate on them using rule-based filters, which are ideal for mobile systems because they do not require n-gram corpora statistics or classification models [11] to be stored on the device. Framework or application developers can then use our rule-based filter nodes or implement their own.

Rule-based filters can operate on structured data coming from multiple social sensors (and potentially named entities from any named entity recognition node). These filters are applied to the data stream in order to prune out less significant or irrelevant data when multiple candidate values are available, where the framework developer, application developer, or the user can configure these rules. For example, if the user wants to meet up with one of several friends who are within a given vicinity, a rule-based filter can prune the candidate list and select one of them. Several types of information (such as contact names, relationships, and shared social groups) can be retrieved from the data provided by the social sensors.

E. Physical and social data fusion

Data can be heuristically fused in different ways [18], and in this section we describe a few basic but useful techniques for fusing the physical and social streams described previously to extract meaningful user context that can be used to build compelling top-level applications.

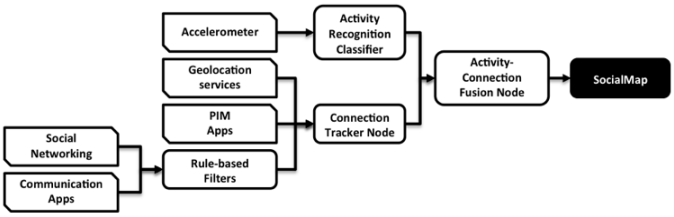

Fig. 6. Graph topology of Social Map.

Fig. 6 shows the topology for our Social Map application. We describe the final user-facing application in the next subsection, but here we discuss its two key data fusion components, Connection Tracker Node and Activity-Connection Fusion Node. These physical-social fusion nodes are representative of what developers would implement: reading upstream data sources, applying filters and other application logic, and propagating events downstream.

Connection Tracker (CT): The goal of this node is to track the user's contacts who are located physically nearby to the user and then emit this information as an event. Tracking is achieved by combining data from the rule-based filters, the address book PIM application, and the geolocation service. Initially, the rule-based filters listen to contact check-in events from SNS apps (such as Facebook) and prune away insignificant or uninteresting events based on any rules set by the developer or the application user. Rules such as "ignore checkins by contacts who are a co-worker" could be applied at this stage. We assume that the relationship of the contact to the user is available from the user's address book and/or SNS friend list. The filter node then forwards details about the contacts who have checked-in (including their name, relationship to the user, and current location) to the CT node, which also receives periodic events about the user's current location from the geolocation service. The CT node then heuristically combines the user and contact location data from these two sources, calculating the physical distance between the user and his checked-in contact using Haversine estimation, and thereby identifies the contacts who are physically located near to the user (for some distance threshold). The final output is in the form of a contactNearBy predicate event that specifies the nearby contact's name, location, distance from the user, and relationship with the user.

Activity-Connection Fusion (ACF): This fusion node combines data from the upstream Connection Tracker and Activity Classifier nodes, filters and prioritizes the fused data based on rules (again, set by the developer or the user), and propagates the result downstream to the end-user application. More specifically, the ACF node listens for contactNearBy predicate events from the CT node and physical activity predicate events (e.g. IsWalking, IsRunning, and IsDriving) from the Activity Classifier node. The ACF node then combines these two data streams, applies filtering rules, and prioritizes the remaining candidates. Rules such as "ignore contactNearBy events if the contact is more than 2 miles away from the user and the user IsWalking" are implemented. The final output of the ACF node is a prioritized list of nearby contacts (including the name and relationship to the user), the contact's distance from the user, and the user's current activity, which are all given to the user-facing application.

F. Applications

To demonstrate the utility of the framework as well as the activity recognition and social sensing library components described earlier, we implemented two user-facing applications that run completely within a mobile WRT.

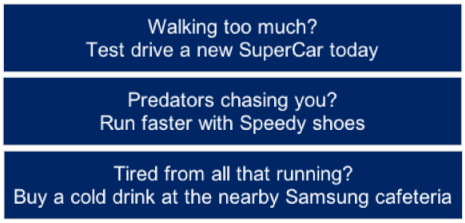

Fig. 7. Mock banner advertisements delivered through ActVertisements. The first and second ads are produced by the IsWalking and IsRunning activity predicates, respectively. The third advertisement is produced by IsRunning and a geolocation predicate.

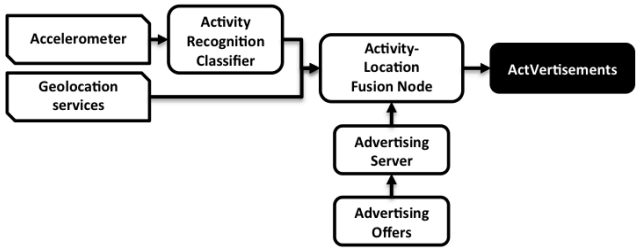

Fig. 8. Graph topology of ActVertisements.

The first application, **ActVertisements**, shows targeted online banner advertisements that are relevant to the user's current physical activity and, optionally, current geolocation. In our system, we created mock advertising offers that are meant to be representative of the type of activity and geolocation-targeted ads that can be shown. In Fig. 7 we show some examples of car, shoe, and food advertisements. To serve these ads, we built the graph topology shown in Fig. 8. We implemented our own lightweight advertising server and executed a simplified bidding system to choose among candidate ad offers, where each offer targets a specific activity and optional geolocation. The ad server receives ad-display opportunities parameterized with the user's (activity, geolocation) pair and then matches the opportunity with available candidate offers. The user's activity, of course, was generated by the Activity Recognition Classifier component in Fig. 8, which corresponds to the library subgraph from Fig. 5. The advertising text is sent to the ActVertisements application, which renders the final banner. In a real

productization, the system would interface with commercial advertising servers that recognize such predicates.

The second application, **Social Map**, shows contextual maps and routes to the user's friends, demonstrating the real-time fusion of physical and social data processing. Fig. 9 shows the user-facing application, where the map in the top panel is generated from Google Maps and is updated to display routes from the user's location to a nearby contact's location, where the highest-priority contact, out of several candidates, was chosen by the Activity-Connection Fusion node (as was described in the previous section and illustrated in the graph topology in Fig. 6). The route in Social Map is shown when that prioritized contact checks into a location. Because this application is meant to be indicative of a product-ready application running on a smartphone, we incorporated the online ActVertisement banner ads as well. Note that the underlying topology only runs one instance of the geolocation services sensor and Activity Recognition Classifier, both of which bifurcate their output to separate ports into the ActVertisements and Social Map graphs.

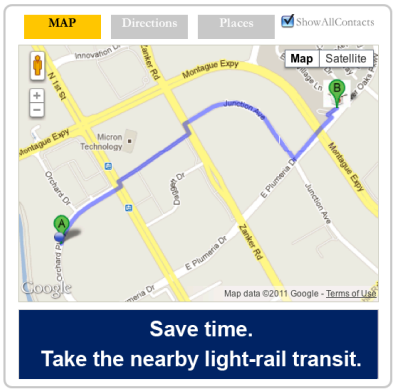

Fig. 9. Screenshot of the Social Map application. The map panel shows a walking route to a contact that was chosen based on sensor fusion.

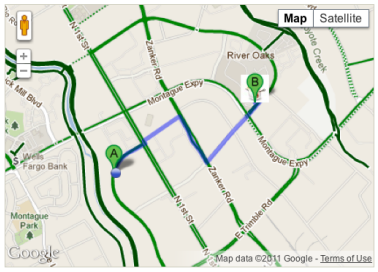

Fig. 10. An alternative route. Note that the previous walking route has been replaced with a driving route due to an IsDriving activity predicate that was recognized by the physical sensors.

We take advantage of the fact that Google Maps offers an API to specify map routes based on means of transportation, and so the resulting type of the route, such as a walking path, running trail, or street, can be based on the user's current detected physical activity, namely IsWalking, IsRunning, or IsDriving, respectively. In Fig. 9, the map shows a walking path from the user to his contact because the Activity Classifier determined that the user was currently walking (perhaps due to the fact that he was taking a stroll when the meeting opportunity became available). Once the user starts driving to the contact, the Activity Classifier determines that he is driving, and the map will be updated with a driving route, shown in Fig. 10, that Google Maps provides for us, presumably with amenable traffic conditions.

IV. Experiments and Results

For the purpose of experimentation, we conducted trials to measure latency and power consumption. Our results show that Web SocialSense and a mobile Web platform show great promise for supporting mobile sensing applications.

A. Configuration

We ran our Web SocialSense applications successfully on:

- Tizen WRT on a Galaxy smartphone
- Mobile Firefox on a Nexus S Android smartphone
- Opera WAC WRT on a Nexus S Android smartphone

We saw little to no difference between the mobile platform implementations since the hardware was similar. Our measurement platform was a Samsung smartphone running Tizen with a 1 GHz single-core ARM CPU, 512 MB of RAM, 16 GB of storage, 802.11 b/g/n, accelerometer, gyroscope, and a 1.5 Ah battery. The OS runs a WebKit-based WRT container that supports W3C-standard applications.

B. Application latency

In Fig. 6 we showed the architecture of our Social Map application. To evaluate its execution, we measured the latency for major components. For the activity classifier node, the measurement is the time to complete classification on one window. For other processor nodes, it is the time that the component receives an input-port event to the time that it emits its output-port event. For a source node, it is exactly the time that the underlying data source is queried to when the node emits an output. Table 1 shows these intra-node latencies in milliseconds. For completeness, we also include the time to initialize the entire graph. Most components completed well within 50 ms on our target platform. The Facebook source node polled the online site through Wi-Fi.

Component	mean	median
Activity classification processor	34.33	34.0
Connection tracker processor	2.0	2.0
Activity-connection fusion	1.0	1.0
Facebook source node (includes networking)	338.22	332.0
Entire graph initialization	111.67	111.0

Table 1. Latencies (in milliseconds) for the Social Map graph components shown in Fig. 6. At least ten measurements were conducted for each one.

C. Power consumption

To evaluate the impact of Web SocialSense on battery use, we ran a series of experiments that measured the power drain of the system using our Samsung smartphones listed previously. We used a Monsoon Solutions Power Monitor hardware meter that logs drawn wattage by directly supplying and measuring power to the smartphone. To isolate the use due to the Web SocialSense framework itself, we zeroed-out the wattage measurements by subtracting away the baseline drawn power that was logged while the smartphone was idle.

In Fig. 11 we show the power consumption response for an accelerometer-sensing application. The Web SocialSense source node polls the accelerometer at 32 Hz and sends the resulting 3-D vector to a sink node, which then renders the result on the webpage. The graph shows modest power usage: after the application starts (after the 2-second mark), the observed power draw has a mean of 317.26 mW and a median of 293.50 mW. These results stand up reasonably well against the 185.18 MW that we measured with an accelerometer-sensing Android application on the same hardware.

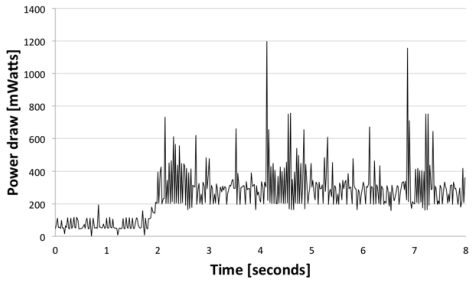

Fig. 11. Power consumption response of a baseline accelerometer-polling Web SocialSense application.

We then conducted a series of experiments that show the power consumption response for varying graph topology lengths and polling rates. The results in Table 2 reinforce the intuition that the power draw increases with more nodes in a serial branch and a higher offered (synthetic) input load in Hz.

Number of serial nodes	Offered input load	Mean power draw	Median power draw
2	100 Hz	125.53 mW	109.12 mW
2	1000 Hz	876.27 mW	875.83 mW
1000	1000 Hz	939.43 mW	932.69 mW

Table 2. Web SocialSense Power consumption response with various configurations of nodes and offered load.

V. Conclusion

We evaluated the viability of implementing sensor processing applications at the Web Runtime layer, a cross-platform approach that obviates the need for developers to re-implement their sensing software using different programming languages and libraries specific to mobile platforms. To that end, we created Web SocialSense, a JavaScript library that provides a graph topology-based framework for developers to implement compelling context-aware applications using sensor acquisition, processing, and fusion from both physical and social software sensors.

Through demonstration and experimentation, we showed that Web SocialSense and a mobile Web platform offer great promise for supporting sensing applications. Specifically, we showed utility at two levels: we first implemented subgraph library components to perform human activity recognition and social sensing and then used them to build two complete end-to-end mobile Web applications, Social Map and ActVertisements. Experiments revealed that the framework is efficient in terms of latency and battery consumption.

In the future we can improve Web Social Sense in several ways. The system can detect cycles when the graph is constructed and finalized. The activity recognition component can be improved to identify more motions. Finally, social sensing can be offloaded to cloud services.

References

[1] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," In Proc. of ACM IMC, 2009.

[2] D. Bannach, P. Lukowicz, and O. Amft. "Rapid Prototyping of Activity Recognition Applications," IEEE Pervasive Computing, vol. 7, no. 2, April-June 2008.

[3] L. Bao and S. Intille. "Activity recognition from user-annotated acceleration data," In Proceedings. of Pervasive, 2004.

[4] A. Chowdhury, B. Falchuk, and A. Misra. "MediAlly: A Provenance-Aware Remote Health Monitoring Middlware," In Proceedings of IEEE PerCom, 2010.

[5] T. Das, P. Mohan, V. Padmanabhan, R. Ramjee, and A. Sharma. "PRISM: Platform for Remote Sensing Using Smartphones," In Proceedings of ACM MobiSys, 2010.

[6] J. Finkel, T. Grenager, and C. Manning. "Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling," In Proceedings of ACL, 2005.

[7] Firefox OS. www.mozilla.org/en-US/firefox/os/

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. "The WEKA Data Mining Software: An Update," SIGKDD Explorations, vol. 11, issue 1.

[9] J. Hicks, N. Ramanathan, D. Kim, M. Monibi, J. Selsky, J. HAnsen, and D. Estrin. "AndWellness: An Open Mobile System for Activity and Experience Sampling," In Proceedings of Wireless Health, 2010.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," In Proceedings. of EuroSys, 2007.

[11] D. Jurafsky and J. Martin. Speech and Language Processing, 2nd edition, Pearson Prentice Hall, 2008.

[12] J. Kwapisz, G. Weiss, and S. Moore. "Activity recognition using cell phone accelerometers," In ACM SensorKDD, 2010.

[13] J. Lester, T. Choudhury, and G. Borriello. "A Practical Approach to Recognizing Physical Activities," In Proc. of Pervasive, 2006.

[14] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. "Energy-Accuracy Aware Localization for Mobile Devices," In Proceedings of ACM MobiSys, 2010.

[15] H. Lu, J. Yang, Z. Liu, N. Lane, T. Choudhury, and A. Campbell. "The Jigsaw Continuous Sensing Engine for Mobile Phone Applications," In Proc. of ACM SenSys, 2010.

[16] C. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval, Cambridge University Press, 2008.

[17] E. Miluzzo, et al. "Sensing meets mobile social networks: the design, implementation, and evaluation of the CenceMe application," In Proceedings of ACM SenSys, 2008.

[18] H. Mitchell. Multi-Sensor Data Fusion, Springer 2007.

[19] Mozilla. "The Baseline Compiler Has Landed," retrieved Oct. 31, 2013. blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/

[20] C. Rogers. "Web Audio API," retrieved December 1, 2011. dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html

[21] Tizen mobile operating system. www.tizen.org

[22] J. White, C. Thompson, H. Turner, B. Dougherty, and D. Schmidt. "WreckWatch: Automatic Traffic Detection and Notification with Smartphones," Mobile Networks and Applications, 16(3), June 2011.

[23] Valve NodeGraph. developer.valvesoftware.com/wiki/Nodegraph