Sentio: Distributed Sensor Virtualization for Mobile Apps

Hillol Debnath, Narain Gehani, Xiaoning Ding, Reza Curtmola and Cristian Borcea
Department of Computer Science, New Jersey Institute of Technology, Newark, NJ
Email:{hd43, narain.gehani, xiaoning.ding, crix, borcea}@njit.edu

***Abstract*—This paper presents Sentio, a distributed middleware designed to provide mobile apps with seamless connectivity to remote sensors when the sensing code and the sensors are not physically on the same device, e.g., when the sensing code is offloaded to the cloud. Sentio presents the apps with virtual sensors that are mapped to remote physical sensors. Virtual sensors can be composed into higher-level sensors, which fuse sensing data from multiple physical sensors. Furthermore, they are mapped to the best available physical sensors when the app starts and re-mapped transparently to other physical sensors at runtime in response to context changes. Sentio was designed to work without modifications to the operating system and to provide low-latency access to remote sensors, which is beneficial to apps with real time-requirements such as mobile games. We have built a prototype of Sentio on Android. We have also developed four apps based on Sentio to understand the programming effort and evaluate the performance. The development of the apps shows that complex sensing tasks can be implemented quickly, benefiting from Sentio's high-level API. The experimental results show that Sentio achieves good real-time performance.**

I. Introduction

Mobile apps, ranging from photo utilities [1], [2] to health monitoring [3] and well-being [4], [5], [6], [7], use sensors to provide context-aware features. Despite their widespread deployment, these apps face challenges in accessing remote sensors in three situations: (1) app components are offloaded to the cloud and need to access sensors on the mobile device where the app was started; (2) an app on a mobile device (e.g., smart phone) needs to access sensors on other personal devices of the user (e.g., smart watch); and (3) an app on a mobile device needs to access sensors on mobile devices belonging to other users.

Computation offloading [8], [9], [10], [11], [12] can enhance the computational and energy resources available to mobiles. However, the computation cannot be readily offloaded if it contains sensor accessing code. The offloaded code will not work unless programmers modify the sensor accessing code and write several other new components. This problem could be solved by a mobile sensing framework that virtualizes sensors and provides apps in the cloud with seamless connectivity to the sensors.

Many new context-aware apps could benefit from a framework that allows them to leverage the collective power of a user's personal sensors. Scenarios where a user carries multiple smart devices (e.g., a smart phone in the pocket, a smart watch on the wrist, and possibly a tablet in the backpack) are becoming common. For instance, a mobile game running on a powerful mobile device (e.g., tablet) may access the accelerometer on a device worn by the player (e.g., smart watch) to offer a more user-friendly and immersive experience. Sharing sensors among users can also be useful in various scenarios (e.g., collaborative sensing apps); sharing policies would need to be defined and enforced for such apps.

Unfortunately, many existing mobile sensing frameworks [13], [14], [15], [16] are typically limited to allowing an app to use sensors only from its device. To allow simple development and deployment of mobile sensing apps that access sensors from multiple devices, programmers should be provided with a high-level and flexible API that works across devices. This API should also allow multiple apps on the same or different devices to access concurrently the same set of remote sensors. However, dealing with remote sensors is challenging because sensors in different devices have different features and different hardware APIs. In addition, certain apps (e.g., mobile gaming) need real-time performance and must execute tasks such as sensor exploration, sensor management, and data aggregation/fusion with low latency.

Some of the above issues have been addressed in the literature, but the proposed solutions [17], [18], [19] lack generality, as they target only a few specific scenarios or particular sensors. Furthermore, none of them provides support for computation offloading, sensor composition, and sensor selection based on user-specified criteria (e.g., real-time operation, power savings).

Another feature not provided by existing solutions for remote sensor access is runtime adaptation to sensor conditions. For example, the same type of sensor can be available to a user from multiple devices (e.g., GPS is available in smart phone, smart watch, and tablet). Even though the sensor type is the same, sensors from different devices have different accuracy, speed, and power characteristics. Therefore, a mobile sensing framework should intelligently choose the "best" sensor among the available sensors of the same type. In addition, context may require an app to switch access from one sensor to another at runtime. For example, the battery status of device hosting the current "best" sensor could be monitored and, in case of significantly low battery, the sensor access should seamlessly be switched to a sensor from another device.

This paper presents Sentio, a distributed middleware, which enables apps to seamlessly access sensors on remote devices. The two types of apps that Sentio supports are: (1) mobile apps that offload components to the cloud; and (2) mobile apps that utilize the collective sensing capabilities of the user's personal sensors. Sentio could be extended to support

apps that access sensors from different users by incorporating privacy/access control policies. Sentio virtualizes the personal sensors of a user and presents a flexible, high-level API to access them. As shown in Figure 1, Sentio presents apps with a high-level abstraction in the form of a Virtual Sensor System (VSS). A VSS is a collection of virtual sensors mapped to physical sensors located on mobile, wearable, and IoT devices. Virtual sensors provide a unified interface to local and remote sensors for app development. Multiple virtual sensors could be leveraged to provide composite virtual sensors, which perform sensor aggregation and fusion. Sentio is also able to select the "best" sensor of a certain type and to dynamically re-map a virtual sensor to a physical sensor at runtime. Since it is implemented at the application level, Sentio is easy to deploy.

We implemented a prototype of Sentio using Android-based smart phones and smart watches. We used Sentio to build two proof-of-concept mobile apps: SentioApp and SentioFit. We also modified two open source games to use Sentio API: Tilt Control and Space Shooter. We ran multiple experiments using these four apps. For example, we ran the Space Shooter game on a mobile device and used a virtual accelerometer mapped to a smart watch as a game controller. Our experimental results demonstrate that Sentio can guarantee real-time delivery of sensor data and impose a minimal overhead.

II. Sentio Overview

Sentio creates a virtual sensor system (VSS) that facilitates app access to physical sensors located on different devices. A virtual sensor behaves as a local and universal sensor abstraction. A virtual sensor has a type (e.g., accelerometer) and can be mapped to physical sensors of the same type located on different devices. Composite sensors can be built on top of several virtual sensors to perform sensor fusion or aggregation. Sentio provides default sensor data fusion methods, but the programmers can specify their own methods. An example of a composite sensor is a "climbing" sensor that combines readings from a heart rate monitoring sensor and a barometric pressure sensor to warn mountain climbers when they should rest or drink more water at high altitude. Sentio uses a heuristic algorithm to select or switch to the "best" available sensor of a given type, as a function of energy consumption, latency, or sensor accuracy.

Sentio also supports sensor access for code offloaded to the cloud. Let us consider an activity recognition task in a smart phone app, which accesses the accelerometer in the phone. When the app offloads the activity recognition task to the cloud, Sentio creates a virtual accelerometer in the cloud and seamlessly maps it to the physical accelerometer of the phone. The offloaded code in the cloud can access the accelerometer sensor in the same way it was doing from the mobile.

Sentio is designed and implemented as a distributed middleware, with components on all personal smart devices of a user. An instance of the middleware on a device manages the physical sensors of the device and executes tasks such as sensor registration, sensor mapping, and data collection. More importantly, each instance is in charge of fulfilling the requests

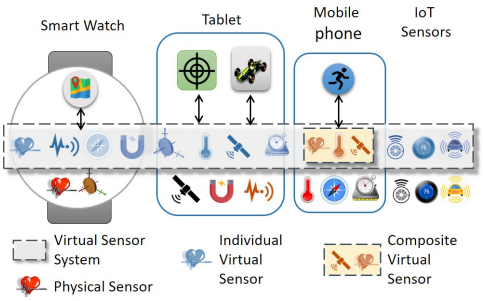


Fig. 1. Sentio creates a virtual sensor system across different devices. A mobile health monitoring app uses Sentio to access a composite sensor built over three virtual sensors that abstract physical sensors: a heart rate monitor in the smart watch, a GPS in the tablet, and a temperature sensor in the phone.

from instances on other devices to retrieve sensor data. Sentio exposes the same high-level API to apps running on any smart device owned by the user; the same API is available for apps offloaded to the cloud.

It is worth mentioning that IoT sensors which are supported by OSs, such as Android Things [20], can run Sentio similarly to smart phones or smart watches. Other IoT sensors can be accessed and managed by Sentio through smart devices using Arduino [21] which connects the sensors via Bluetooth.

Sentio uses a computing entity (CE) [10] residing in the cloud for sensor management and for enhancing the computational, power, and storage resources available to mobiles. Each user has one CE, which can be a virtual machine or a container. The CE maintains the VSS sensor registry and the current states of all virtual sensors exposed by Sentio to apps. In addition, the CE is used to provide support for computation offloading to the cloud. App code offloaded to the cloud can access virtual sensors through a Sentio instance running in the CE.

In Sentio, one device (generally, the smart phone) is considered the primary device and all the other devices are considered as secondary devices. If the CE becomes unreachable due to network problems, the primary device uses ad hoc networking to perform VSS management. To be able to do this switch, CE and the primary device periodically synchronize the VSS state with each other.

Sentio does not employ ad hoc networking by default because it includes sensors that are not in the proximity of the user. For example, smart home apps running on user's mobile can access home sensors from anywhere.

III. Related Work

A. System Support for Accessing Remote Sensors

Accessing remote sensors is partially addressed in Beetle [17], Rio [18], and BraceForce [19], which target a few specific scenarios. Beetle aims to control Bluetooth Low Energy (BLE) peripherals; Rio focuses on sharing I/O devices; BraceForce targets the development of remote sensing apps. Sentio, on the other hand, aims to provide a general solution that allows any mobile app to seamlessly access a wide range of remote sensors (e.g., sensors on other mobile devices, IoT sensors). Table I illustrates the novelty and benefits of Sentio when compared with the solutions mentioned above.

TABLE I
COMPARISON BETWEEN SENTIO AND RELATED WORKS. P MEANS PARTIAL SUPPORT.

	RIO	Beetle	BraceForce	Sentio
Deployable with unmodified OS	✗	✗	✓	✓
Code offloading support	✗	✗	✗	✓
Sensor composition support	✗	✗	✗	✓
Sensor selection optimization	✗	✗	✗	✓
Support for more than two devices	✗	P	✓	✓
Dynamic sensor switching support	P	✗	✗	✓
High level API	✗	✓	✓	✓
Support for BLE peripherals	✗	✓	✗	✗
Concurrent sensor access	✗	✓	✓	✓
Works with heterogeneous systems	✓	✗	✓	✓

Unlike Sentio, the comparison systems do not provide support for code offloading, sensor composition, and sensor selection based on user-specified criteria (e.g., real-time operation, power savings). In addition, Beetle and Rio do not work with unmodified operating systems (i.e., they need rooted mobile devices). Beetle also does not work for non-BLE sensors and multiple smart devices trying to access sensors from each other because Android devices cannot work as BLE peripherals. Similarly, RIO does not support multi-device setup. Braceforce, on the other hand, works with unmodified operating systems and multi-device setup. However, it does not support dynamic sensor switching at runtime, and it does not work with Android Wear OS.

B. Programming Abstractions for Remote Sensing

SeeMon [22] is a framework for developing context-aware applications, which monitors user context through sensors. Similar to Sentio, SeeMon provides sensor control policies. However, it does not provide a general-purpose interface or support for accessing sensors across devices or fusing data from multiple sensors. MobileHub [23] is a system that rewrites applications to leverage a sensor hub for power efficiency. Unlike Sentio, MobileHub only supports access to local sensors through the sensor hub.

HomeOS [24] provides a peripheral abstraction to connect home devices (TV, printer, lights) to a PC. However, its centralized setting lacks flexibility for our use cases. Sentio employs a distributed setup of mobile devices where multiple apps can run on multiple devices and can control sensors on other devices. Furthermore, HomeOS is not designed to run on mobile devices, as it needs a dedicated PC. It also does not provide support for features offered by Sentio such as sensor-code offloading, sensor fusion, selecting the best sensor, and seamless switching of sensors due to context changes.

BOSS [25] provides an API for sensors/actuators placed in a building to simplify application development. However, programmers need to write device/sensor-specific code (e.g., $set\_min\_airflow$). Sentio, on the other hand provides a high-level API, designed specifically for mobile apps. BOSS also needs to run several services (Hardware Presentation Layer, Transaction Manager) on a dedicated computer collocated with BMS. Thus, it cannot run on a setup with only mobile devices.

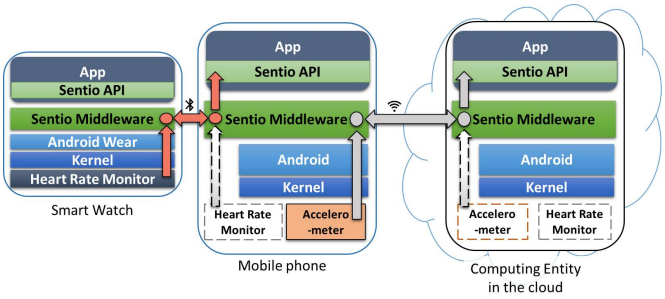


Fig. 2. Sentio provides a unified view of the VSS via Sentio API. Solid boxes show physical sensors; dotted boxes show virtual sensors.

IV. SENTIO DESIGN

As shown in Figure 2, Sentio consists of a set of middleware instances running on smart devices and the cloud. The virtual sensors created on a device and the physical sensors embedded in the device are managed by its local middleware instance. The access to virtual sensors is also handled by the middleware instance, which either forwards the requests to the corresponding physical sensors if they are located on the device or forwards them to the middleware instances on the devices where the physical sensors are located.

A. Sentio API

Sentio API follows an event-driven and callback-based asynchronous design. For each Sentio API call, a program needs to provide a callback function. The API call sends a request to the Sentio middleware about the desired operation, and returns immediately when the request is sent. The middleware handles the request and returns the necessary data to the app by invoking the callback function. This asynchronous API design fits well with the architecture of Sentio, in which apps and the Sentio middleware run in different processes and exchange information using events and messages.

```
1 SensorListener listener = new SensorListener() {
2   @Override
3   public void onSensorDataAvailable(
        SensorDataBundle bundle) {
4     //TODO: process sensor data
5   }
6 };
```

Listing 1. Code for a sensor listener.

As shown in Table II, Sentio API methods are mainly designed to 1) query about available sensors in the VSS, 2) specify callback functions for receiving data from the middleware, 3) create virtual sensors (individual or composite), and 4) register and unregister sensors.

An app starts by querying Sentio for the list of available sensors in VSS. The example in Listing 1 explains how a program accesses a virtual sensor. Specifically, a program must create an instance of *SensorListener* callback interface (line 1) and implement the *onSensorDataAvailable()* callback function in the instance (line 3). After the program registers for the virtual sensor with the *SensorListener* instance, the Sentio middleware will automatically pass sensor data as *SensorDataBundle* objects to the callback function. Information about the

TABLE II
LIST OF SENTIO API METHODS

API	Description
getAvailableSensorList(SensorListListener listListener)	Fetches a list of available sensors in the sensor registry. The list is received by the callback function of the "listListener" interface.
registerSensor(SentioSensor sensor, SensorListener listener)	Registers the "sensor" from the VSS. Sensor data will be received by the "listener".
unregisterSensor(SentioSensor sensor)	Unregisters the "sensor".
SensorListListener callback interface	
onSensorListAvailable(Map<SentioDevice, List<SensorType≫ sensors)	Receives a mapping between "SentioDevice" and its sensors from the middleware.
SensorListener callback interface	
onSensorDataAvailable(SensorDataBundle data)	Receives sensor data from the middleware.
FuseAction interface (used for writing custom fusing functions)	
SensorDataBundle fuseData (SensorDataBundle... data)	An interface method for programmers writing custom fusing functions. It receives SensorDataBundle objects from the sensors in the current composite sensor, applies the fusing function to the received data, returns a single SensorDataBundle object.
SentioSensor builder API	
addSensor(SentioDevice device, SensorType type)	Adds a sensor of type "type" from the device "device" to the virtual sensor.
addSensor(SentioSensor sensor)	Adds the virtual (composite/individual) sensor "sensor" to the current virtual sensor.
addSensor(SensorType type, SensingMode mode)	Adds the sensor selected based on the "type" parameter (i.e., sensor type) and the "mode" parameter (i.e, POWER_SAVING, REAL_TIME, or ACCURACY).
samplingRate(SamplingRate rate)	Defines the sampling rate or frequency (NORMAL, UI, GAME, or FASTEST).
addListener(SensorListener listener)	Specifies the listener which will receive the sensor data.
fuse(FusingMode mode)	Specifies the fusing mode (Combine, Average, Competitive, or Complementary).
fuse(FuseAction fuseAction)	Specifies a customized fuse function.

sensor (e.g., source device, sensor type, etc) is also included in the *SensorDataBundle* object.

```
1 SentioSensor sensor = SentioSensor.newBuilder()
2       .addSensor(SensorType.MAGNETIC_FIELD,
            SensingMode.ACCURACY)
3       .addSensor(SensorType.GPS, SensingMode.
            POWER_SAVING)
4       .samplingRate(SamplingRate.NORMAL)
5       .addListener(listener)
6       .fuse(FusingMode.COMBINE)
7       .build();
```

Listing 2. Code for building a composite sensor.

The example in Listing 2 shows how to build a composite sensor. This composite sensor is for environment monitoring. It consists of a magnetic field sensor in the ACCURACY sensing mode (added on line 2) and a GPS sensor in the $POWER\_SAVING$ sensing mode (added on line 3). Since our sensor building API follows the Java builder design pattern [26], programmers can add an arbitrary number of sensors by repeatedly calling *addSensor()*. An individual virtual sensor can be built by calling *addSensor()* only once. A polymorphic version of *addSensor()* can be leveraged to build a composite sensor from already existing virtual sensors.

The sensing modes specified in *addSensor()* calls can direct Sentio to select the appropriate sensors for the request. If a program does not specify a mode, Sentio will balance accuracy, speed, and energy consumption and will select sensors based on a method described in Section IV-B1. Line 4 specifies the sampling rate of the virtual sensor. *SamplingRate* is an *enum* with four possible predefined values, similar to the ones used in Android: NORMAL, UI, GAME, FASTEST. The first three represents 200 ms, 60 ms, and 20ms sampling periods, respectively. FASTEST directs the middleware to sample data as frequently as possible. Line 5 specifies the *listener* callback for receiving sensor data, and line 6 specifies the fuse function to be used for fusing data. A few predefined fusing functions are provided by Sentio, such as Combine, which combines data points together and delivers the results in an array. Programmers can also apply a custom fuse function by implementing the *FuseAction* callback interface.

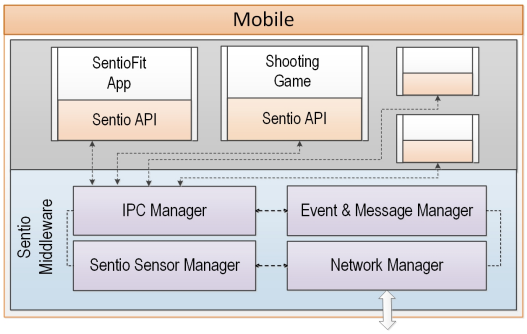


Fig. 3. The Sentio middleware architecture.

B. Sentio Middleware Design

The Sentio middleware instance on each device has four major components, as shown in Figure 3. **IPCManager** manages the inter-process communication (IPC) between Sentio apps and the middleware instance. An IPC channel is established when an app initiates a SentioManager object, and is released when the app stops and unbinds it from the middleware. The execution of the Sentio middleware is driven by events and messages. Every Sentio API call is translated to a corresponding low level event, and sensor data is sent in messages. The **Event and Message Management** component queues events and messages after they are received and ensures that they are dispatched later. The **NetworkManager** handles communication between the middleware instances on different devices. By default, Sentio uses the Internet. However, if

devices are disconnected from the Internet but are physically located in proximity, the NetworkManager works in a P2P fashion using techniques such as WiFi-direct or Bluetooth.

As the core of Sentio **SentioSensorManager** manages the local physical and virtual sensors on a device. When Sentio is initialized on a new device, this manager runs a sensor discovery process to create a local sensor registry (LSR), which includes the specifications of the sensors, such as precision/resolution, the lowest and highest sampling rates, and power consumption rating. It then sends the LSR along with the device ID to the cloud entity, where a global sensor registry (GSR) is formed. The GSR will be used to find appropriate sensors to build virtual sensors. The SentioSensorManager at the cloud entity manages the GSR and is responsible for discovering and managing IoT sensors located in the user environment. As described in Section II, we assume that IoT sensors are accessed through a smart device running Sentio. Based on the user's location, her cloud entity interacts with cloud entities of nearby smart devices to discover IoT sensors.

When a request is received to register a virtual sensor, SentioSensorManager identifies the physical sensors that meet the required criteria and map them to the virtual sensor. It does so by searching the GSR fetched from the cloud. To accelerate searching, a copy of the GSR is maintained on each device and is synchronized with the cloud. If the desired sensors are located on other devices, the local SentioSensorManager will contact the corresponding SentioSensorManagers on those devices in order to get access to those sensors. It will also forward the mapping information to these SentioSensorManagers, which will be used to deliver sensor data to the requesting app.

The SentioSensorManager also monitors context changes in the device, such as battery level and network connection changes. It notifies the SentioSensorManagers of remote devices about the status changes when necessary (i.e., remote devices have virtual sensors based on physical sensors on this device). As a result, remote devices may switch the physical sensors used for the virtual sensors.

1) Sensor Selection: When there are multiple sensors available for building a virtual sensor, the SentioSensorManager needs to select a sensor. If a sensing mode is specified by the program (Section IV-A), it makes the selection based on the sensing mode. Specifically, if the sensing mode is $REAL\_TIME$, Sentio considers the highest sampling frequency that the sensor can support, as well as the latency for the sensing data to arrive to the app. If the sensing mode is $POWER\_SAVING$, Sentio considers the energy consumption of the sensor, the energy for transferring the sensing data, and the battery level of the host device of the sensor, in order to avoid selecting a sensor on a device with a low battery level. If the sensing mode is $ACCURACY$, Sentio picks the sensor with the highest resolution.

$$score = w_a * \frac{1}{precision} + w_d * \frac{1}{min_{delay} + latency_{comm}} + w_p * \frac{battery_d}{power_s + power_{comm}} \quad (1)$$

If the program does not specify a sensing mode, Sentio makes the selection by balancing speed, accuracy, and power saving. It computes a score for each sensor using Equation 1, and selects the sensor with the highest score. In Equation 1, $precision$ is the smallest change a sensor can detect. min_{delay} and $latency_{comm}$ refer to the supported minimum sampling period of the sensor and the communication latency to receive data from the sensor; $power_s$ indicates the power consumption rating of a sensor; and $battery_d$ indicates the remaining battery of the host device. $power_{comm}$ refers to the power consumption for transporting data from this sensor to the requesting device. w_a, w_d, and w_p are the weights assigned to the accuracy, delay, and power consumption factors, respectively. In our implementation, we give equal weights to these factors.

2) Sensor Data Rate Control: Physical sensors may not be able to report data at steady rates, which are required by many apps. To ensure that virtual sensors report data at steady rates, besides common mechanisms, such as filtering, Sentio also uses a rate controller, which buffers data points arriving early and uses extrapolation to project data points that arrive late.

3) Managing Composite Sensors: To build a composite sensor, the SentioSensorManager first creates component virtual sensors. Then, it establishes and maintains the mapping between the composite sensor and the component virtual sensors. A buffer is used to stage the data from component virtual sensors, since they may have different data rates. Every time when enough data (one sample from each component virtual sensor) is accumulated to calculate a fused data point, the fusing function is invoked. In case of missing data points from a sensor within the required time window (defined by the requested sampling rate), Sentio extrapolates a value based on the previous data points. Since the data rate of a composite sensor is affected by the data rate of each component virtual sensor, it can be more unsteady than that of an individual virtual sensor. Therefore, even if we have applied data rate control to each component virtual sensor, we still use a data rate controller to regulate the fused data points to further stabilize the rate.

V. Sentio Implementation

We have implemented a prototype of Sentio on standard Android (for mobiles, tablets, and the cloud entity) and Android Wear (for smart watches). The prototype Sentio SDK has 3,127 lines of code (LoC). The mobile module of Sentio (i.e., standard Android OS) has 3,726 LoC, and the wearable module (i.e., Android Wear OS) has 561 additional LoC. The cloud entity is an Android x86 virtual machine hosted and managed by the Avatar system [10]. Although the implementation is Android-based, the implementation techniques are generic and can be used in implementations on other systems.

The implementation follows a Message-Oriented Middleware architecture, where different components communicate through events and messages. Since apps and Sentio middleware instances are different processes, we use Android's binder IPC interface for the communication among them. Each Sentio middleware instance contains a queuing mechanism

to manage events and messages, using the observer design pattern (dispatchers observe on queues). The NetworkManager uses kryonet TCP library [27] in mobile modules and MessageApi[28] provided by Google Play Services in wearable modules for the communication between the Sentio middleware instances on different devices. MessageAPI uses BlueTooth and WiFi for low level communication.

The Sentio middleware uses Android's sensor SDK to access and manage physical sensors. To respond to the status changes of the devices, we use Android's BroadcastReceiver mechanism to monitor status changes (e.g., $BATTERY\_LOW$) and register the intended actions, which are invoked by the Android automatically on status changes.

Sentio supports concurrent access from multiple apps to the same physical sensor. This is achieved by carefully designing the data structures and routing to consider concurrent accesses. The paper does not elaborate this aspect because Sentio mainly focuses on providing a personal sensing eco-system, where accessing a sensor concurrently from multiple apps is rare or does not make sense (e.g., accessing the same accelerometer to control two games).

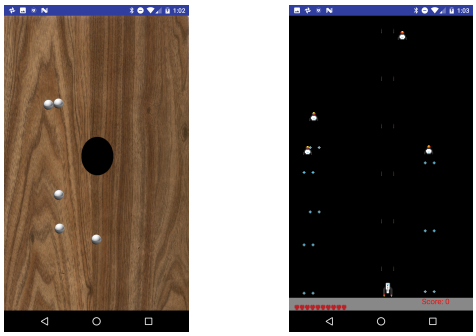
VI. Application Case Studies

We validated Sentio with 4 mobile apps: two proof-of-concept apps developed from scratch (i.e., SentioApp and SentioFit) and two open source mobile games, adapted to work with Sentio (i.e., Tilt Control and Space Shooting).

SentioApp is built to verify that various types of virtual sensors can be built with sensors on different types of devices by calling Sentio APIs. The app first visually shows the sensors available in the VSS. Then, it allows users to read any of these sensors and build composite sensors by selecting individual sensors. Once a virtual sensor is built and registered, the app shows the sensed data from this sensor on the screen. We have run this app on various combinations of smart devices. For example, we ran the Sentio middleware on several phones (Nexus 6, Nexus 5X, Moto X) and a watch (Samsung Gear Live) to form the VSS, and ran the app on Nexus 6.

SentioFit is built to test composite sensors, particularly the support for custom fusion functions. It uses a composite sensor built from a heart rate monitor(HRM), a step detector, and a barometric pressure sensor. A custom fusion function is used to i) check the heart rate, ii) keep track of the number of steps per minute, and iii) convert air pressure to altitude. Based on these data, the fusion function generates the final result: KEEP_GOING, SLOW_DOWN, RUN_FASTER. The app uses the result to train the user, while ensuring his heart rate is not too high. We ran this app on a Nexus 5X. The sensors selected during the execution were located in a Samsung Gear Live watch (HRM) and the Nexus 5X (barometric pressure sensor and step detector). The test shows that programmers can build composite sensors easily without worrying about data collection, communication, and aggregation.

To demonstrate that real-time apps can easily use Sentio to access sensors, we selected two open source mobile games from Github. Figure 4 shows the screenshots of these games.



(a) Tilt control (b) Space shooting

Fig. 4. Open source mobile games modified to work with Sentio

In the Tilt Control game, a user tilts the mobile to put all the balls in the central circle. In the Space Shooting game, a user tilts the mobile to control the space ship. Thus, both games need to use accelerometers. We modified the sensor-accessing code in the games, such that, when they are running on one device (e.g., a tablet with a big screen), they can access its local accelerometer or the accelerometer on another device (e.g., a light-weighted phone used as the game controller). We tested the games by running the apps on a Nexus 5X phone and using the Nexus 5X phone, a Samsung Gear Live watch, or a Nexus 6 phone as controller. We repeated the tests for two Android's sampling rates, $GAME$ and $FASTEST$.

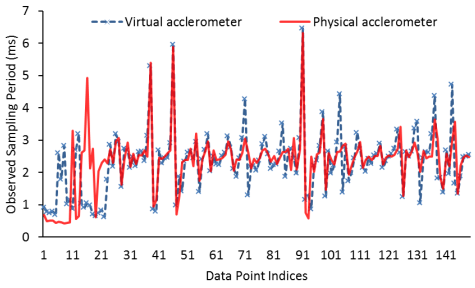
The gaming experience was smooth and without any lag for all the scenarios. Developing and running these games demonstrate the simplicity and robustness of Sentio API. It takes only 6 lines of code in Java (excluding comments and without wrapping the lines for readability) for each game to allow it to build and access a virtual accelerometer, including the sensor listener interface (similar to code listing 1) and the code building the virtual sensor (similar to code listing 2).

VII. Performance Evaluation

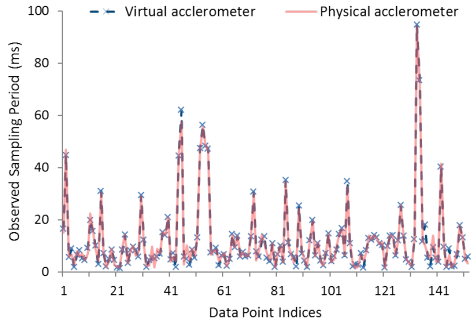
Our evaluation is mainly to assess the impact of Sentio on sensor data delivery, with a special focus on the performance achieved by apps that have real-time constraints (e.g., mobile games). In addition, we evaluated the costs of API calls, sensor composition, and sensor switching. We have used Android-based smart phones, tablets, and smart watches, as well as virtual machines (VMs) running in an OpenStack-based cloud. Each VM runs Android 6.0 x86 64-bit OS and has 4 virtual CPUs and 3GB of memory. Mobiles communicate with the cloud using a secure WiFi network. The communication between phones is done through WiFi; the communication between a phone and a watch is done through Bluetooth.

A. Real-time performance of virtual accelerometers

To verify that Sentio does not impose significant latency on sensing data as observed by the apps, particularly real-time apps, we ran several experiments with the Space Shooting game accessing a virtual accelerometer at a requested rate. We measure the observed sampling period of the virtual accelerometer in two scenarios: (1) the physical accelerometer



(a) Physical accelerometer on phone



(b) Physical accelerometer on watch

Fig. 5. Comparison between the observed sampling period of a virtual accelerometer and a physical accelerometer when the physical accelerometer is on the phone and on the watch. The app runs on the phone and the sampling rate is set to FASTEST

is on the phone; (2) the physical accelerometer is on the smart watch. In both scenarios, the game app runs on the phone. Figure 5 shows the results. We measured the observed sampling period of the virtual sensor and actual sampling period of the physical sensor. The variation between the two curves is due to the overhead introduced by Sentio. This metric does not include the network delay, but it includes the jitter. Similar to real-time multimedia streaming, Sentio will introduce a one time network delay. Then, the apps will be influenced only by the jitter, which is captured by our metric.

We first used Android's sampling rate "$FASTEST$" to verify how Sentio performs under the most constrained real-time demands. This sampling rate requires to retrieve and deliver data as fast as the sensor hardware supports it. The difference between the average observed sampling period is 27μs when the physical sensor is on the phone and 10μs when the physical sensor is on the watch. We observe a higher variability for the observed sampling rate of the physical sensor on the watch, and this translates in a similar variability for the virtual sensor.

These results demonstrate that Sentio works well even under strict real-time constraints and imposes minimal overhead on virtual sensor access latency. The main factors that influence this latency are the physical sensor access handled by OS and the network latency. Thus, Sentio is expected to scale well, as long as the host OS provides fast access to physical sensors and the network is not congested.

We then varied the sampling rate to see its impact on Sentio's performance. Table III shows the results, where two phones communicated using WIFI. We observe that the

TABLE III
OBSERVED SAMPLING PERIOD (IN MS) FOR DIFFERENT SAMPLING RATES WHEN USING WIFI COMMUNICATION. THE GAME RUNS ON ONE PHONE, AND THE PHYSICAL SENSOR IS ON THE OTHER PHONE.

	$NORMAL$		UI		$GAME$		$FASTEST$	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Avg	199.48	199.39	39.82	39.69	19.16	19.09	10.4	3.98
StDev	43.7	4.47	21.42	1.86	77.01	1.76	8.22	0.7
Median	199	200	39	40	12	19	9	4

TABLE IV
OBSERVED SAMPLING PERIOD (IN MS) FOR DIFFERENT SENSOR TYPES WHEN SENTIOAPP RUNS ON ONE PHONE, AND THE PHYSICAL SENSOR IS ON THE OTHER PHONE. WIFI IS USED FOR COMMUNICATION. $NORMAL$ SAMPLING RATE.

	Light		Magnetic Field		Gyroscope		Orientation		Pressure	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Average	395.5	395	200.6	195.9	203.7	201	194.7	194.3	198.1	196.2
StDev	518.2	430.4	63.7	10.5	33	6.2	49.9	1.1	99.6	144.4
Median	197	200	200	196	199	201	191	194	247	185

difference in average observed sampling rate for the virtual sensor and the physical sensor is very low (<1ms) for GAME, UI, and NORMAL sampling rates. For the FASTEST sampling rate, the difference is 6.6 ms. These differences are slightly larger than in the case of Bluetooth communication. We attribute this result to the contention in our public WiFi network, which leads to queuing effects in Sentio, especially for high sampling rates. Nevertheless, the results demonstrate that Sentio also works well with WiFi communication.

B. Performance for other sensors

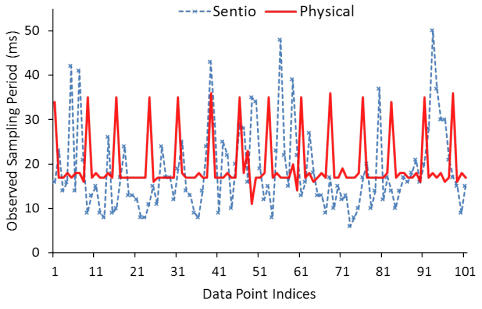
We have shown that Sentio works efficiently for real-time apps (e.g., games) using the accelerometer for the experiments, because this is the sensor most commonly used in games. Next, we evaluate performance of Sentio for other types of sensors, such as light, magnetic field, gyroscope, barometric pressure sensor, etc. For this experiment, we have used SentioApp running on a Nexus 6 phone. The app uses various virtual sensors, which are mapped to sensors from a Moto X phone. The $NORMAL$ sampling rate (5 samples per second) is used. As shown in Table IV, the maximum difference in the observed average sampling period is 4.69 ms. This is a small value when compared to the expected period of 200 ms. Therefore, Sentio is expected to work efficiently for all the sensors we tested. The standard deviations with light and pressure sensors are higher than those with other sensors because they are designed to generate data only upon state changes instead of providing continuous measurement steadily.

C. Performance for offloaded code

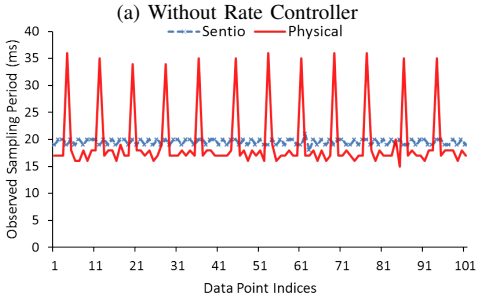
Sentio simplifies code offloading because the offloaded code to the cloud can seamlessly access sensors on mobiles. We evaluated how Sentio performs once the code is offloaded using the Tilt Control game. The game is hosted in the cloud, and it uses a virtual accelerometer mapped to Moto X's accelerometer. As shown in Table V, the highest difference in the average observed sampling period (5.32 ms) is observed for the $FASTEST$ sampling rate. Sentio worked very well for

TABLE V
OBSERVED SAMPLING PERIOD (IN MS) WHEN THE TILT CONTROL GAME IS OFFLOADED TO THE CLOUD FOR DIFFERENT SAMPLING RATES.

	NORMAL		*UI*		*GAME*		*FASTEST*	
	Virt	Phy	Virt	Phy	Virt	Phy	Virt	Phy
Average	200.7	200.5	46.79	46.73	19.18	19.13	9.32	4
StDev	8.12	0.99	12.28	9.9	5.39	0.93	3.44	0.81
Median	201	201	43	40	19	19	9	4



(a) Without Rate Controller



(b) With Rate Controller

Fig. 6. Observed sampling period comparison for accelerometer with and without the rate controller. The sampling rate is $GAME$

other sampling rates. These results prove that code offloading with Sentio is practical for context-aware apps.

D. Jitter reduction using a rate controller

The results we presented so far use sensor data as soon as they become available. As virtual sensors may be mapped to different devices, network delay will cause data to arrive at the app with irregular frequency. This explains the zig-zag pattern in some of the previous graphs. Sentio uses a rate controller to reduce this type of jitter and smoothen the sensor data over time. Figure 6 shows the benefits from the rate controller for SentioApp running on Nexus 6 and accessing Moto X's accelerometer. The $GAME$ sampling rate is used. We observe that the jitter is substantially reduced. Similar results, omitted for brevity, have been obtained for virtual magnetic field sensor and the $NORMAL$ sampling rate.

E. Seamless switching of sensors during context changes

Sentio switches sensors seamlessly during context changes on the devices embedding the sensors. For example, if the battery on a device is low, Sentio re-assigns the physical mapping of sensors from this device to another device. We have measured the latency experienced by the app during this re-mapping. We have run SentioApp in Nexus 6 using a virtual magnetic field sensor, which is physically mapped to

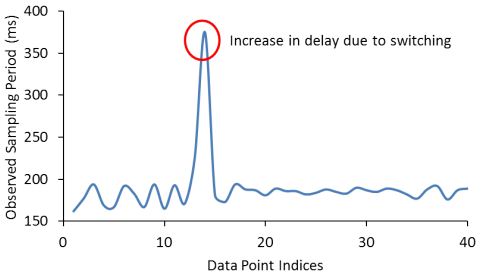


Fig. 7. Delay observed by SentioApp during a sensor switching from a watch to the local phone. NORMAL sampling rate is used.

the magnetic field sensor of the smart watch. In our scenario, Sentio senses when the battery level in the smart watch drops below a specific threshold and re-maps the physical mapping of Nexus 6's virtual sensor to the physical magnetic field sensor of the local device (i.e., Nexus 6). The latency for this re-mapping consists of several components: the delay to pass an event from the smart watch to Nexus 6, with the low-battery context change information; the delay to find another suitable sensor from the registry; and the delay to register the newly chosen sensor and receive data from it.

Fig 7 shows that sensor switching causes a clear increase in the observed sampling rate: 375 ms as opposed to the average value of 184 ms before and after the switch. We note that this increase only affects the immediately next data point after switching. For all practical purposes, skipping one data point is an acceptable compromise.

VIII. CONCLUSION

This paper presented Sentio, a middleware that virtualizes sensors for mobile apps. Sentio provides a unified view of a personal sensing ecosystem. Apps can use Sentio API to access any virtual sensor in real-time, no matter on which device its associated physical sensor is located. They can also selectively access the most suitable sensor of a particular type and can build composite virtual sensors using the same API. The Sentio middleware can transparently re-map virtual sensors to physical sensors in response to context changes. We have implemented a prototype of Sentio and two proof-of-concept apps. In addition, we adapted two open source apps to work with Sentio. We evaluated Sentio using these four apps, and the results show that Sentio does not introduce significant overhead in sensing. Therefore, it can be used for many types of context-aware apps, including apps that have tight real-time constraints such as mobile games.

IX. ACKNOWLEDGMENT

This research was supported by the NSF under Grants No. CNS 1409523, SHF 1617749, and DGE 1565478, and by DARPA/AFRL under Contract No. A8650-15-C-7521. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DARPA, and AFRL.

REFERENCES

[1] H. Debnath and C. Borcea, "Tagpix: Automatic real-time landscape photo tagging for smartphones," in *Proceedings of the International Conference on MOBILe Wireless MiddleWARE, Operating Systems and Applications (Mobilware)*, Nov 2013.

[2] C. Qin, X. Bao, R. Roy Choudhury, and S. Nelakuditi, "Tagsense: A smartphone-based approach to automatic image tagging," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM, 2011.

[3] J. L. Carús Candás, V. Peláez, G. López, M. A. Fernández, E. Álvarez, and G. Díaz, "An automatic data mining method to detect abnormal human behaviour using physical activity measurements," *Pervasive Mob. Comput.*, vol. 15, no. C, pp. 228–241, Dec. 2014.

[4] M. Rabbi, M. H. Aung, M. Zhang, and T. Choudhury, "Mybehavior: Automatic personalized health feedback from user behaviors and preferences using smartphones," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. ACM, 2015.

[5] N. D. Lane, M. Lin, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell, and T. Choudhury, "Bewell: Sensing sleep, physical activities and social interactions to promote wellbeing," *Mobile Networks and Applications*, 2014.

[6] H. Lu, D. Frauendorfer, M. Rabbi, M. S. Mast, G. T. Chittaranjan, A. T. Campbell, D. Gatica-Perez, and T. Choudhury, "Stresssense: Detecting stress in unconstrained acoustic environments using smartphones," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, ser. UbiComp '12. ACM, 2012.

[7] *Google Fit*, https://www.google.com/fit/ (Accessed:March-10-2017).

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010.

[9] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proceedings IEEE INFOCOM*, March 2012.

[10] C. Borcea, X. Ding, N. Gehani, R. Curtmola, M. A. Khan, and H. Debnath, "Avatar: Mobile distributed computing in the cloud," in *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, March 2015.

[11] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013.

[12] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, Apr. 2010.

[13] *Intel Context Sensing SDK*, https://software.intel.com/en-us/context-sensing-sdk/details (Accessed:March-10-2017).

[14] *Google: Android Sensing SDK*, https://developer.android.com/guide/topics/sensors/sensors_overview.html (Accessed:March-10-2017).

[15] W. Brunette, R. Sodt, R. Chaudhri, M. Goel, M. Falcone, J. Van Orden, and G. Borriello, "Open data kit sensors: A sensor integration framework for android at the application-level," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012.

[16] G. Cardone, A. Cirri, A. Corradi, L. Foschini, and D. Maio, "Msf: An efficient mobile phone sensing framework," *International Journal of Distributed Sensor Networks*, vol. 9, no. 3, p. 538937, 2013.

[17] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein, "Beetle: Flexible communication for bluetooth low energy," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. ACM, 2016.

[18] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: A system solution for sharing i/o between mobile systems," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. ACM, 2014.

[19] X. Zheng, D. E. Perry, and C. Julien, "Braceforce: A middleware to enable sensing integration in mobile applications for novice programmers," in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft 2014. New York, NY, USA: ACM, 2014.

[20] *Android Things*, https://developer.android.com/things/get-started/index.html (Accessed:March-10-2017).

[21] *Android to Arduino interface using Bluetooth*, http://www.instructables.com/id/How-control-arduino-board-using-an-android-phone-a/ (Accessed:March-10-2017).

[22] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008.

[23] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall, "Enhancing mobile apps to use sensor hubs without programmer effort," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. New York, NY, USA: ACM, 2015.

[24] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An operating system for the home," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, 2012.

[25] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler, "Boss: Building operating system services," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. USENIX Association, 2013.

[26] J. Bloch, *Effective Java*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[27] *Kryonet*, https://github.com/EsotericSoftware/kryonet (Accessed:March-10-2017).

[28] *Google Play Services - MessageAPI*, https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi (Accessed:March-10-2017).