BraceForce: A Middleware Enabling Novice Programmers to Integrate Sensing in CPS Applications

Xi Zheng, Dewayne E. Perry, Christine Julien
(jameszhengxi@utexas.edu, perry@mail.utexas.edu, c.julien@utexas.edu)

The Center for Advanced Research in Software Engineering
Mobile and Pervasive Computing Lab, Electrical and Computer Engineering
The University of Texas at Austin

TR-ARiSE-2013-003

BraceForce: A Middleware Enabling Novice Programmers to Integrate Sensing in CPS Applications

Xi Zheng, Dewayne E. Perry, Christine Julien
The Center for Advanced Research in Software Engineering
Mobile and Pervasive Computing Lab
The University of Texas at Austin
jameszhengxi@utexas.edu,perry@mail.utexas.edu,c.julien@utexas.edu

ABSTRACT

Even as the use of sensor networks to support CPS applications grows, our ability to seamlessly and efficiently incorporate sensor network capabilities remains astoundingly difficult. Today, accessing remote sensing data and integrating this data into the adaptive behavior of a dynamic user-facing CPS application requires interacting with multiple sensor platform languages, data formats, and communication channels and paradigms. We present BraceForce, an open and extensible middleware that allows developers to access the myriad remote sensing capabilities inherent to cyber-physical systems using very minimal code. Further, BraceForce incorporates event- and model-driven data acquisition as first-class concepts to provide efficient access to sensing while retaining expressiveness and flexibility for applications. This paper presents the BraceForce middleware architecture and key abstractions, describes their implementations, and provides an empirical study using BraceForce to support CPS applications.

Keywords

sensor drivers, sensor network, mobile computing, smartphones, cyber physical systems

1. INTRODUCTION

Developing and deploying CPS applications involves a large amount of low-level programming that requires interacting with different (often proprietary) data formats, languages, and operating systems. In practice, applications are built for specific sensor network platforms with little potential for portability to other platforms or integration with other sensors. Debugging CPS applications that inherently integrate with a physical environment requires not only the aforementioned integration of the application with sensing but also the use of a testing harness that, at debugging time, accesses sensed data about the physical environment for the purposes of validating the actions of the application. Integrating sensing for application and debugging support in a way that is easy, flexible, and portable is essential for supporting CPS application development.

In general, research in this space has been focused on demonstrating the feasibility of applications, the development of support services such as routing protocols or energy-saving algorithms, or on advancement of hardware platforms and operating systems. Little focus has been applied to effective development support for applications that integrate the capabilities of networked sensing platforms in easy to use and extensible ways. In addition to the variety of data formats, communication technologies, and programming platforms a developer must tackle, CPS applications also require handling network dynamics and energy constraints.

This paper introduces BraceForce, a middleware for CPS application development that simplifies the development, deployment, and debugging of CPS applications. In supporting application deployment, BraceForce allows the developer to connect the application to sensing assets in the deployment environment. In supporting application debugging, BraceForce can be used to monitor a test environment using capabilities that may not be available in the deployment environment. Architecturally, BraceForce defines functional *tiers* that encapsulate related aspects but coordinate in such a way that the tiers' deployment to particular physical assets is flexible. Different tiers can be deployed on user-facing devices or on sensing devices with limited capabilities, directly addressing and leveraging the specific capabilities and intentions of each device. BraceForce provides auto-discovery of new sensing and computing assets, allowing easy integration of new capabilities into existing applications or automatic and seamless extension of a debugging environment. As BraceForce discovers new components, the tiered architecture organically extends to these new components, flexibly deploying the necessary tiers to the new components.

Instead of using a proprietary programming language (e.g., nesC [9]), supporting a single hardware platform (e.g., ODK [5]), or creating a new standard (e.g., Dandelion [19] or DASH [7]), BraceForce presents a simple Java programming interface for integrating new sensing capabilities. To add a new sensor to BraceForce, a developer just needs to implement a simple sensor driver interface that provides essential commands (e.g., `open`, `close`, `query`) and configuration. The driver implementation can be specific to the particular sensor but basically translates raw data into meaningful BraceForce sensor data constructs, guided by the provided BraceForce interfaces. BraceForce handles issues such as thread management, networking, remote procedure call, etc.

BraceForce supports both the traditional *pull* and more flexible *push* based interaction with sensing devices. Out of concern of energy-saving, BraceForce's default form of interaction is *event-driven data acquisition*, in which sensed data is only pushed to an interested application on the occurrence of an *event*. Applications specify thresholds on changes in sensed values that determine when new samples are returned. Such an approach has been shown to dramatically reduce energy costs of interacting with networked sensors. Existing approaches that support some integration of sensing capabilities with mobile application development [5] provide only continuous sampling or instantaneous polling, which require the application to tune the interaction with the sensing platform and therefore either expend extraneous energy or miss important sensed value changes.

BraceForce embraces *model-driven data acquisition* (MDDA) [14, 21, 22, 24, 27] to reduce energy costs of integrating sensing in CPS applications. MDDA suppresses sensor readings that are *predictable* according to some *a priori* or learned model. BraceForce supports (*i*) temporal models of data evolution based on previous sensor readings [24]; (*ii*) models based on underlying physics principles of the sensed system [21]; (*iii*) models derived from applying data mining to prior sensed data [14]; and (*iv*) models expressing correlations of data in space and time [27]. While the use of model-driven data acquisition is itself not novel to BraceForce, what is new is how BraceForce integrates model-driven data acquisition in a general purpose programming framework for acquiring and interacting with sensed data.

This paper describes BraceForce, from its flexible architecture to a prototype implementation. We directly assess BraceForce's ability to ease CPS application development through an exploratory study of BraceForce applied to real CPS applications that rely on networked sensors. BraceForce provides a clear design contract for integrating new sensors into existing applications. This paper makes the concrete contributions in three areas:

1. *Supporting CPS application developers and users*
   - BraceForce provides a simple and unified programming interface to reduce programming barriers of CPS applications that integrate sensing.
   - BraceForce's tiered architecture enables dynamic updates to CPS applications and their physical deployments without intervention by the users.
2. *Supporting flexible and expressive CPS applications*
   - BraceForce can be deployed to heterogeneous devices in different configurations to meet constraints of CPS applications and devices.
   - BraceForce supports both pull- and push-based interaction with sensing devices.
   - BraceForce embraces model-driven data acquisition to address energy concerns of CPS applications.
3. *Studying CPS application development*
   - We demonstrate and evaluate how BraceForce makes both the development process more approachable and the resulting CPS application more efficient.
   - We demonstrate that MDDA within BraceForce can significantly reduce the network overhead, which in turn reduces energy consumption.

The next section places this paper's contributions in the context of related work. In Section 3, we provide complete details of the BraceForce middleware. Section 4 presents the empirical design for our evaluation, while Section 5 describes its results.

2. RELATED WORK

BraceForce aims to make sensor programming transparent to CPS application developers by explicitly and intentionally hiding low-level sensing concerns. BraceForce provides CPS application developers dynamic deployment and automatic discovery of networked sensors, distributed data caching and aggregation capabilities, and presents the networked sensors through a combination of event-driven and model-driven accessors, providing both flexibility for developers and efficiency for the deployment. Existing approaches tackle similar and in some cases overlapping concerns, but, to our knowledge, BraceForce is the first to provide transparent support of CPS application development and debugging.

Sensor Platforms. Much work on sensor networks has focused on hardware and software platforms to support increasingly sophisticated capabilities. TinyOS [13] is an operating system for sensor networks that enables developers to use networked sensors to solve distributed sensing, computational, and coordinating tasks. Its focus has largely been on supporting increasingly complex and sophisticated applications at the expense of usability and flexibility [17]. The Robot Operating System (ROS) [23] provides a component-oriented style of programming, in which components communicate via publish/subscribe mechanisms and two-way service calls, both using user-defined *topics*. However, ROS only supports a communication scale of one machine, and connection mechanisms are among ROS components running on the same machine; CPS systems require coordination among distributed components. ROS also comes with a non-trivial learning curve.

Arduino has gained wide popularity due to simplicity, high degrees of usability, and the resultant enabling of rapid prototyping. While programming for Arduino is more straightforward, programmers are still required to interact with the Arduino at a very low-level of abstraction. This does not allow for flexible updating and discovery of local sensing devices. The Android Sensor Framework [1] provides a programming interface to access hardware and software sensing components on Android. This framework is also limited in its abstract capabilities; the developer is still entirely responsible for thread control and other essential concerns for accessing sensors. Further, the Android Sensor Framework does not enable connecting to external sensors connected to the device via BlueTooth, USB or other connectivity modes. Instead, the developer must use other libraries.

Integrating sensing in CPS applications requires high level abstractions to provide automatic sensor discovery and flexible and efficient access to the sensor data via event- and model-driven data acquisition. BraceForce unifies access to a variety of sensor platforms and provides a flexible and expressive application programming interface with high-level programming constructs tailored to networked sensor integration.

Sensor Programming Frameworks. Easing programming of sensor networks has received significant attention. Maté [18], for example, is a tiny virtual machine that allows developers to concisely express sensor programs and cause these programs to be dynamically deployed. A virtual machine approach is very flexible, but Maté comes with the cost of increased complexity due to the severe and unmitigatable resource constraints of the target environment [12]. The mixed (and richer) capabilities of our target CPS environment allow us to circumvent these resource constraints using a distributed architecture. In addition, we can embed data-driven techniques in our programming abstractions to make acquiring and integrating sensor data a natural part of the programming task.

SensorWare [4] shares the resources of a single node among many applications. SensorWare's primary abstractions have a network focus and are intuitive for sensor and networking experts but do not promote data and data integration for application developers. MiLAN [11] builds on networking and discovery protocols, using a plug-in mechanism to incorporate arbitrary protocols. Application developers use QoS graphs to specify their sensing requirements, and MiLAN uses this information along with the sensor network state to determine how to configure the network and sensors to meet the provided requirements. This high-level data-centric abstraction is the style we target in our middleware. We couple these abstractions with automatic sensor discovery and a distributed architecture that, by its nature, addresses the mixed resource constraints of our target environment.

Dandelion [19] supports developing wireless body sensor applications on smartphones using a programming abstraction called a *senselet*, which abstracts a device driver and allows applications to integrate data from that device through remote method invocation. This abstraction is data-centric and provides a jumping-off point for BraceForce's abstractions, but Dandelion does not incorporate higher-level programming constructs for aggregation, model-driven data acquisition, and automatic sensor discovery and integration.

Perhaps most similar to our goals, Open Data Kit (ODK) Sensors [5] simplifies deployment of smartphone applications that rely on data from a smartphone's external sensors (e.g., connected via USB and BlueTooth). The application logic may rely on data from both on board and external sensors, and the sensor driver developer implements specific driver interfaces for configuring sensors, packaging the data they generate, and connecting that data into the Android framework. Interactions with sensors are confined to pull-based acquisition with only locally connected sensors. ODK Sensors is tightly integrated with Android, limiting its applicability.

We focus on easing the integration of distributed sensor data into CPS applications without limiting the expressiveness of sensing capabilities or over-utilizing precious constrained resources. This demands not only automatic sensor discovery and integration but also abstractions for acquiring sensor data in ways other than through direct sensor polling.

Model Driven Data Acquisition. Model-driven data acquisition (MDDA) can limit costly communication with networked sensors by suppressing polling and notifications from the sensors except when the sensor readings deviate from some pre-defined or learned model. Gupta et al. studied the problem of collecting spatially correlated data in a wireless sensor network, based on the theory of dominating sets [10]. Other work has focused on achieving data suppression using temporal and spatial data correlation either by dividing the network into clusters [20], exploiting domain knowledge regarding reasonable ranges of sensed values [26], or combining temporal and spatial correlations [27]. Other approaches use simple models of sensed value trends to generate readings only when the sensed value deviates from the model's predicted value. Simple linear models are very effective [24], and the approach can also be applied to in-network aggregation of raw values [8].

We are motivated to use similar techniques in BraceForce to suppress data without sacrificing the quality of knowledge about the sensed entity. BraceForce allows CPS developers to incorporate different models of MDDA both at the level of individual sensors and at the level of an entire distributed application to handle different deployment scenarios and meet specific energy requirements of CPS applications. We avoid approaches that demand a particular topological structure (e.g., designated clusters of nodes) to avoid unnecessary rigidity in a dynamic network of supporting sensors and rely on simple models that have shown significant gains.

3. BRACEFORCE

We first describe BraceForce from an abstract architectural perspective; we then detail our prototype implementation.

3.1 BraceForce Abstract Architecture

BraceForce comprises five abstraction layers: the *sensor driver* layer, the *data* layer, the *distribution* layer, the *distributed data cache and aggregator* layer, and the *CPS application* layer. Fig. 1 shows one example BraceForce deployment with these five layers.

3.1.1 BraceForce Layered Architecture.

Sensor Driver Layer. In the current state of the art, CPS application developers must understand low-level protocols and hardware-specific aspects of sensors to be able to write sensor drivers. Moreover, the raw data from the sensor has no open standard and often loses the original temporal information associated with the sensor data retrieval. These issues demonstrate a gap between the low-level sensing capabilities of both on-board and external sensors and the application space. At BraceForce's base, the sensor driver layer bridges this gap. The sensor driver layer encapsulates functions related to interacting with a sensor, from configuration, through starting, querying, reconfiguring, stopping, and cleaning up allocated resources. The sensor driver layer forces driver developers to adhere to a design contract enabling these common functionalities and unifying on-board and external sensors into a shared data packet format that can be used throughout the remainder of the BraceForce architecture. A driver developer creates the connection between the sensor and the driver layer by defining how raw sensor data (e.g., in the form of a binary array) is converted into a BraceForce *sensor data response*, which defines an abstract data type unified across all sensor data. Each sensor data response maps a data type (or types) to a value (or values) and associates a timestamp with the data. This timestamp ensures that data used by higher layers is *fresh* within the requirements of the application.

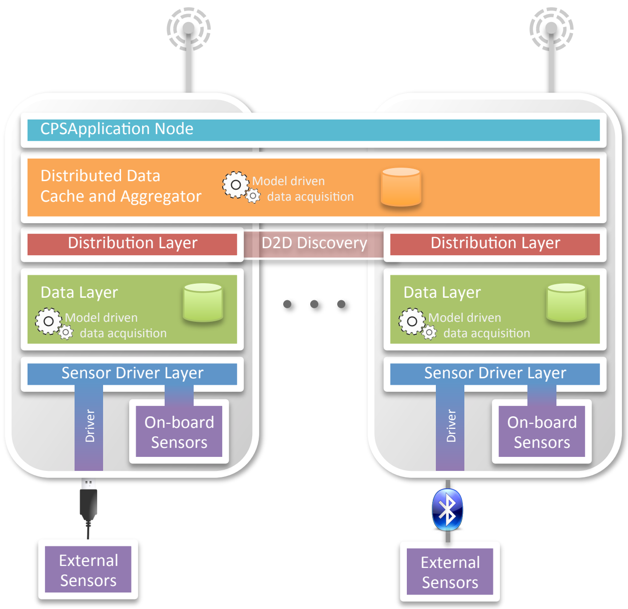

Figure 1: BraceForce: Uniform Environment

Data Layer. Mobile operating systems allow sensors to be accessed directly using BlueTooth, Near Field Communication (NFC), USB, and proprietary interfaces to on-board sensors. This manner of access is device-specific and requires a non-trivial amount of low-level development and testing using interfaces that are not portable across platforms. These programming methods also usually entail a steep learning curve. BraceForce encapsulates these communication and sensor framework interactions in the data layer, thus explicitly separating user-application code above from sensor- and platform-specific code below. The data layer also controls data retrieval for locally connected sensors (both on-board and external), where the options include *push*, in which a sensor pushes every data reading to the data layer; *pull*, in which the data layer periodically polls the connected sensor; and *event-driven*, in which a sensor notifies the data layer only upon the occurrence of some predefined "event. " At the data layer, BraceForce introduces single-device MDDA, whereby simple models of temporal correlation can be used to suppress sensor readings as long as they follow an expected model [24].

Distribution Layer. CPS applications require a distributed view of active and available sensors, both on the local device and available remotely on other devices in the network. Programmers must use message passing mechanisms or RPC-like connectors to make devices in the network communicate and coordinate their actions. Developers must also handle location transparency and other intrinsic issues related to distributed programming (e.g., concurrency, failures, and time synchronization). In the distribution layer, BraceForce provides automatic discovery of other BraceForce-enabled devices, and thereby other BraceForce connectable sensors. BraceForce uses a combination of best effort protocols for the on-the-fly discovery and reliable protocols for initiating and accepting remote method calls to transfer sensor data from discovered sensors. The distribution layer also extends the data acquisition modalities of the data layer by defining data listeners that bridge between the data layer below and the aggregation layer above and, more importantly, across distributed devices. One specific function of this bridge is to use guidance from the cross-device MDDA performed at the aggregation layer (described next) to correctly configure and query underlying (distributed) sensors. Within the distribution layer, BraceForce's automatic detection of distributed sensors can be deployed alongside the Sensor Driver Layer, circumventing the data layer, simplifying the BraceForce deployment on devices that have significant computation and storage constraints.

Distributed Data Cache and Aggregator. CPS applications that require large scale deployments often include sensing devices that have severe computation and storage constraints. In this case, it is highly ideal that data storage and processing are done in backend servers or other more powerful devices in the network. Programming these features in a scalable and reliable way is a challenge for CPS application developers. BraceForce's aggregation layer is specifically designed to deal with this challenge through a unified view of all of the dynamic sensing capabilities of the networked sensing support infrastructure. As new sensors are discovered on-the-fly or known sensors disappear, these dynamics are handled seamlessly by BraceForce. Given a distributed view of aggregate sensing capabilities, the aggregation layer can perform high-level MDDA, for example, by using more sophisticated models based on spatial correlations or learned relationships among sensed data [22]. In our prototype, we demonstrate the potential for this high-level MDDA using a spatial correlation model that suppresses sensor readings that are similar to neighboring readings. More generally, from the aggregation layer, BraceForce can expressively direct data acquisition (e.g., by choosing between event-driven data acquisition, the push modality, and the pull modality) for all of the sensors under its purview. This layer also encapsulates data mining and compression.

CPS Application Node. To support the CPS application node, BraceForce maintains a registry of aggregation and sensing capabilities, and the programming interface allows the application to subscribe to them. Based on the particular application goals, these sensors can be integrated just as part of a testing harness at debugging time or can provide essential functional information for the application at deployment time.

3.1.2 BraceForce Deployment Scenarios.

Fig. 1 showed just one possible deployment scenario in which all of the devices are homogeneous (e.g., smart phones) and have the same capabilities. In BraceForce, different physical assets can support different pieces of the architecture, depending on a particular device's capabilities and functional requirements. In any deployment, the sensor driver layer and the discovery layer must be present on any sensing-capable device, as they are essential to connecting to and getting data from sensors. On moderately capable devices, the addition of the data layer adds multiple modalities for driving data acquisition and the potential for expressive temporal-based MDDA. Fig. 2 shows an alternative deployment in which the system consists of heterogeneous devices; in this example, the application runs on a dedicated high-powered device (e.g., a laptop) that connects to a variety of sensing devices of different capabilities, held together by the distribution layer.

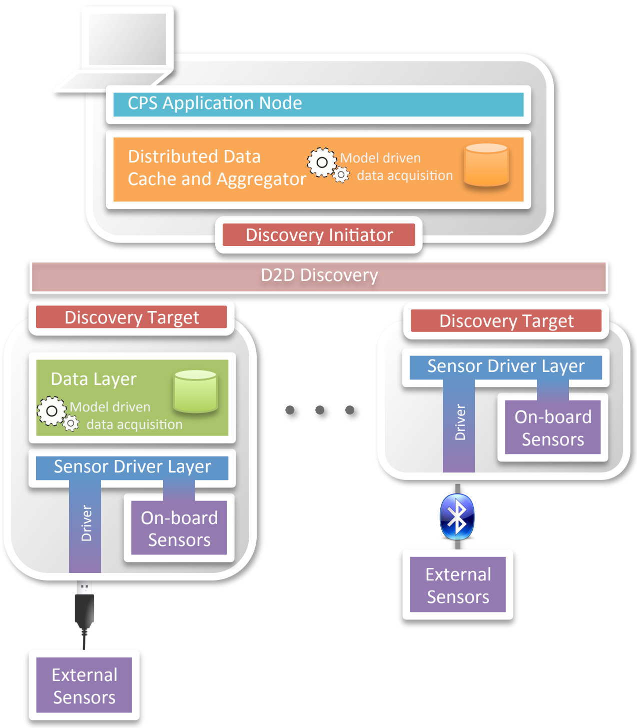

Figure 2: BraceForce: Mixed Environment

We envision three primary use cases for BraceForce. In the obvious case, BraceForce is an integral part of the CPS application. Take a smart home application in which a home controller connects to various sensing devices integrated with the home to control the ambient environment. In this case, the homeowner may introduce and remove components from the home over its lifetime, and, using BraceForce, the changing capabilities are seamlessly reflected to the application.

As a second case, we envision the middleware to be deployed for large scale ad-hoc sensor networks (e.g., to monitor the fire situation in a forest). The Sensor Driver Layer and Discovery Layer can be combined into a service to be deployed on cheap sensor devices that have basic sensing and networking capacity without additional computation and storage resources. These devices can be deployed at a mass scale. The Distributed Data Cache and Aggregator can be deployed as service to more powerful Android devices or a cluster of backend servers for data processing. The sensor data then will be passed to the CPS application node, which runs on user-facing Android devices to notify users (e.g., fire workers and rescuers) of the real time situation in the environment and allow them to make informed actions.

As a third use case, consider a mobile autonomous robot application wrapped in a test harness that connects to expressive sensing capabilities available in the debugging environment but not expected to be available in the deployment environment. At deployment time, the robot may be placed in an unknown territory and expected to perform some tasks. At debugging time, however, the developer may control the environment and may be able to monitor various aspects of the robot using sensors embedded in the environment (e.g., overhead cameras). The debugging program that surrounds the actual control application can use BraceForce to access these sensors for debugging; at deployment time, these connections to BraceForce and the debugging environment are removed.

3.2 BraceForce Implementation

To demonstrate and evaluate BraceForce, we have built the entire architecture in the Android operating system. We choose Android as our initial platform as it is open source and has extensive support for background processes, including several built-in constructs for inter-application communication [25]. BraceForce is not particular to Android, and we avoid using low-level constructs and interfaces specific to Android and not replicated on other platforms.

3.2.1 Android Programming Idiosyncrasies

A handful of accidental complexities arise from our choice of Android; below, we describe these challenges and our solutions as they arise. In addition, to understand the discussions of the architecture, we briefly review a vocabulary related to Android:

Intent: a passive data structure holding an abstract description of an operation to be performed; something that has happened and is being announced.

Android Interface Definition Language (AIDL): allows definition of the programming interface that the client and service use to communicate with each other using interprocess communication (IPC).

Bundle: a data structure of key-value mappings but not limited to a single String/Object mapping.

Android does not provide significant high-level abstractions for programming coordination among distributed devices. From a model perspective, BraceForce assumes such capabilities, e.g., in the form of Java RMI. To support our BraceForce prototype on Android, we therefore implement our own version of RMI by building on several open source projects. We use JsonBeans [15] to serialize and deserialize Java object graphs to and from JSON [6]. JsonBeans is an obvious choice for this task because it is very lightweight (45KB) with no external dependencies. We use ASM [2, 16] to dynamically generate classes involved in the RMI process in binary form.

3.2.2 Unified Data and Subscription Interfaces

Programming for Android requires interacting with the Dalvik virtual machine, and it is inevitable that our implementation accesses some proprietary Android constructs (e.g., Bundle). To remain as general as possible, we implement a wrapper that encapsulates these proprietary components and presents a generic interface to the BraceForce implementation. Within this wrapper, BraceForce translates Android data structures to reusable and portable Java data structures.

Android provides sensor data subscription only for internal sensors. To subscribe for sensor data from external sensors (e.g., sensors connected via USB), developers have to create a subscription model themselves (e.g., using the observer design pattern) and write a large amount of low-level code to access sensor data. Retrieving data from other devices is even more difficult. In BraceForce, we provide a unified subscription interface for developers to access internal sensors, external sensors, and networked sensors. Fig. 3 shows the BraceForce API used to subscribe sensed data from an internal accelerometer (line 1), a temperature sensor (Dallas DS18B20) connected via USB (line 2), and all accelerometers on networked devices (line 3). Line 4 shows how to retrieve sensor data; it is the same regardless of the type of sensor connection. The retrieved data contains a timestamp of the data and where the data is from; data values are accessed through meaningful keys instead of array indices provided by Android.

3.2.3 Thread Management and Android services

Android provides much support for multi-threaded applications. As with any multithreaded environment,

```
1 BraceSensorManager.subscribeSensorEvent(
      BuiltInSensorType.ACCELEROMETER.name(), this,
      this);
2 BraceSensorManager.subscribeSensorEvent("DS18B20",
      this, this, SensorAutoDetection.USB );
3 BraceSensorManager.subscribeSensorEvent(
      BuiltInSensorType.ACCELEROMETER.name(), this,
      this, true);
4 BraceSensorManager.retrieveSensorData(event);
```

Figure 3: Interface for data subscription

the added flexibility comes with a significant increase in complexity. In Android applications that interface with sensing, developers must implement (and debug) threads to listen for and connect to multiple connections. These tasks are far from trivial, as they require the developer to have a deep understanding of the relationships between the operating system and the life cycles of application components. As our user study (described in Section 5) demonstrates, developers have a difficult time navigating the complexities of the Android APIs related to thread management and concurrency.

BraceForce exposes thread managers that explicitly enable thread-safe access to the shared sensor data available to the CPS applications. To enable communication beyond shared memory, we expose the threads and their embodied data using the Android Binder service's remote procedure call capabilities. To conserve energy, BraceForce activates threads only when necessary. This is guided by developers through the interface in Fig. 3, e.g., by specifying the types of communication interfaces to attach to event listeners. For example, in Fig. 3, line 2 specifically indicates that BraceForce should activate the USB thread to communicate with the connected temperature sensor. Alternatively, a developer with less experience in low-level details can indicate, via a boolean parameter, activation of all the networking threads, albeit at increased cost.

3.2.4 Networking

Our prototype supports simple device-to-device discovery based on UDP. The relevant threads for discovery and exposed Android Binder services are built in the distribution layer. When each discoverable service is started within BraceForce, the distribution layer implementation creates a new Android intent and attaches the discovery capability to that intent. When the discovery layer starts a service, it listens on the UDP port specified in its intent and reacts to discovery requests received on this port. The discovery layer also actively engages in discovery by sending UDP packets to neighboring devices. Each node maintains an active list of other nodes; this list is maintained by removing any nodes that have failed to respond to three consecutive periodic requests. Using UDP for discovery is a simple approach to enable automatic device discovery. BraceForce could potentially integrate other on-the-fly discovery mechanisms in the distribution layer, e.g., [3], so that devices can talk to each other even when they are in different networks (e.g., behind NAT).

3.2.5 Sensor Driver Definition and Discovery

BraceForce's driver layer separates CPS application developers from sensor driver developers by providing a contract for implementing new sensor drivers. Fig. 4 shows this contract and the simple `SensorDataResponse` structure, which is mapped to a Java `Hashtable`. The interface shown in Fig. 4 provides a high-level abstraction that allows access to sensor data in key-value pair fashion instead of requiring the application developer to directly interact with myriad forms of raw sensor data.

```
public interface Driver {
  byte[] getCmdForSensorConfig(String configKey,
      Hashtable configParams);
  byte[] getCmdForSensorData();
  byte[] getCmdForStartSensor();
  byte[] getCmdForStopSensor();
  SensorDataResponse getSensorData(List<
      SensorDataPacket> rawData);
  byte[] sendDataToSensor(Hashtable actuatorData);
  void shutdown();
}
public interface SensorDataResponse {
  List<Hashtable> getSensorDataInCollection();
}
```

Figure 4: Interface for defining sensor drivers

The driver interface is exposed through an AIDL file. We have built the drivers for common built-in sensors on Android devices. For any external or uncommon sensors, the sensor driver developer must implement the interface in Fig. 4 to provide access to those sensors[1] . This is accomplished by creating an Android library project that includes the interface implementation and `AndroidDeviceDrivers.aidl`. The project can be uploaded to the mobile devices connected to the sensor(s) for which the driver(s) are written. Given the project's manifest file, the BraceForce driver layer can automatically detect the driver definition and create AIDL stub clients that connect to the new driver service.

The BraceForce data layer automatically discovers connected sensors (e.g., those attached via USB and BlueTooth in our prototype). Once the data layer in BraceForce detects external sensors, a manager agent interacts with the underlying communication channel manager and the created driver stub to capture raw sensor data, convert it to the BraceForce `SensorDataResponse`, and make it available for the higher layers of the architecture. The manager's interface allows the higher level (and ultimately the application developer) to spec-

[1]While the byte[] data type clearly requires programming at a very low-level, the sensor driver developer is an expert in the sensor and, as such, is required to think about sensors at a low-level. The CPS application developer, however, is shielded from these low-level concerns by the high-level data types that BraceForce provides.

ify how BraceForce should interact with the sensor. In the case of push-based interactions or event-driven data acquisition, the application developer defines the behavior of a listener, as shown in Fig. 5.

```
public interface SensorDataChangeListener {
  void bindDataProvider(SensorDataProvider
       provider);
  void onDataChanged(SensorDataChangeEvent event);
}
public interface SensorDataProvider {
  List<Hashtable> getSensorData();
  void addSensorData(Hashtable sensordata);
  void addSensorData(List<Hashtable>
       sensordataList);
}
```

Figure 5: Data Change Listener Interface

The first method in the interface binds the listener to a data provider; the latter provides a wrapper for a specific sensor/driver pair on a specific device. The second method is invoked automatically by the data layer when a reading is pushed or an event notification occurs.

3.2.6 *Model Driven Data Acquisition*

To help CPS application developers to design energy-efficient solutions, BraceForce enables MDDA within the data layer for single device models and within the aggregation layer for distributed models.

In our prototyped data layer, we use the temporal model from [8]; specifically, if the value change for a single sensor between two consecutive readings lies below a specified threshold, the reading is suppressed (Fig. 6). This is a simple form of data suppression based on temporal data correlation; even only slightly more complex models [24] may substantially further reduce the overhead of high quality sensing since the energy of communication dominates computation [27]. Our goal with this simple implementation is to demonstrate how MDDA dovetails with the BraceForce architecture.

```
float lastData =
 (Float)historicalDataPack.get(mainParameterName);
float thisData =
 (Float)currentDataPack.get(mainParameterName);
long accessTime =
 (Long)historicalDataPack.get("timestamp");
long currentTime =
 (Long)currentDataPack.get("timestamp");
if(currentTim-accessTime<timeDelta){
  if(Math.abs(thisData-lastData)<dataDelta){
    Log.d("SensorEventDrivenListener",
          "Data suppressed");
    return;
  }
}
```

Figure 6: Simple temporal data correlation

In the aggregation layer, we apply MDDA using simple spatial data correlation. The data aggregation service maintains a list of nearby devices and their sensing capabilities. Periodically (as specified by the application), BraceForce aggregates sensed values by type from all connected devices. Our model assumes that each sensed value has within its `SensorDataResponse` a location and timestamp. Our implementation of spatially correlated MDDA checks the sensed values from devices located close to one another and suppresses readings from sensors that would be redundant with respect to the spatial coverage of the sensing task. Because the aggregation layer is above the distribution layer in the BraceForce architecture, this data suppression decision must be transmitted to the distributed sensing devices; this is accomplished through the RMI implementation encapsulated within the distribution layer. We use a simple model that suppresses readings for a device physically located between two similarly capable devices if the two devices on either side sense values within a specified threshold of each other.

4. EMPIRICIAL DESIGN

Using our prototype implementation, we have carried out an empirical evaluation to answer the following questions about the performance and use of BraceForce.

Question 1: Does BraceForce simplify development of applications that require interaction with distributed and aggregated CPS sensor data? If so, how and why?

Question 2: What are the potential ramifications of using BraceForce to interface with the sensors?

Question 3: Can BraceForce save energy by employing even primitive predictive models of temporal and spatial data correlation? If so, under what conditions does it work and at what cost to quality of knowledge supplied to the application?

Question 1: Simplifying the Programming Task. We conduct a user study involving twelve participants from a diverse student population and with varying levels of programming experience. All have basic knowledge of Java. We ask each participant to fill in the sensor data retrieval sections for three different applications.[2] Each participant performed each of the three tasks once using BraceForce and once using the Android SDK alone. Each participant was awarded a $20 Amazon gift card. The order of the two subtasks was randomized for each user and each application.

- ◦ Application 1 relies only on a single Android internal sensor (the accelerometer). The application mimics part of a smarthome CPS application that uses an Android device to detect the user's shake orientation. The hardware used is an Android phone.
- ◦ Application 2 relies on an external temperature sensor. The application represents part of a CPS application that requires getting information from external

[2]Videos of how BraceForce works for these three application are available at `http://goo.gl/62WMyc`, `http://goo.gl/KVDQZT`, and `http://goo.gl/5mE64m`

sensors to an Android device. The hardware includes a temperature sensor (Dallas DS18B20) connected via an Arduino MEGA board, which in turn is connected to an Android tablet.

- Application 3 mimics a large scale wireless sensor network. Sensor readings come from accelerometers on low-budget Android phones. The sensor data must be aggregated to a more powerful data processing device (an Android tablet). The sensor data is then displayed in an application running on a user's Android phone. The participants create the sensor retrieval and aggregation components (including a primitive MDDA model of temporal data correlation for which pseudo-code is provided). These components are deployed to the Android mobile phone and an Android tablet, respectively. Connectivity among the devices relies on a local wireless network.

We provide training sessions for the participants, which includes a five minute session on BraceForce, a five minute session on the scope and functions of the applications,[3] and a thirty minute session for the Android architecture and APIs related to sensor data retrieval.[4] The participants were given additional material on Android (e.g., URLs for highly relevant Android programming) to review at home prior to the study. We record for each participant how many hours they spent on Android reading after the training but before the study. To rule out other confounding variables (e.g., human fatigue), we set the time limit for each user study to four hours.

Before the user study, each participant fills out a pre-questionnaire to provide basic information of their programming experience in general, and specifically with Android and with sensors. After each user study, each participant fills out a post-questionnaire about how they feel about using BraceForce compared with Android SDK for each application and their overall feeling about the middleware. If a user fails to complete an evaluation application, in the post-questionnaire we ask what led to the failure and the user's estimate of how many more hours would be needed to complete it.

We measure the development time and accuracy for each user for each of the evaluation applications. The results are averaged across participants. We also provide in-depth qualitative analysis of participants' feedback.

Question 2: Potential ramifications of using BraceForce. To answer the second question, we use the same evaluation applications in the first question. In this case, we implement the Android versions ourselves as the baseline to compare with the BraceForce versions. We measure the accuracy of the applications (using out-of-band validation, for example, by measuring the temperature using an ordinary thermometer) and the running time required to acquire the sensed data; these results are averaged across five runs of each application. In Application 1 and 3, a "run" is defined as 10 distinct rotation tests; in Application 2, a "run" is defined as 10 measurements of body temperature.

[3]The tutorials for both BraceForce and evaluation applications are available at `http://goo.gl/mMXQj5`

[4]Tutorial for Android are available at `http://goo.gl/8j9tBA`

Question 3: Energy Savings with Model-Driven Data Acquisition. To answer the third question, we created a fourth evaluation application, which is a piece of a smart home application in which a user retrieves light readings from multiple sensors deployed around a home. The deployment is at the home of one of the authors; the lighting levels are subject to controlled lighting as well as uncontrolled conditions that include the glow of street lights, passing cars, daylight, etc. The deployment consists of six sensing devices, a pair of devices for data aggregation, and a completely separate laptop that runs the CPS application node.

We employ MDDA at both the data and aggregation layers. We deploy the six sensing devices in pairs; one of each pair executes MDDA, while the second of the pair does not. Across all runs, we measure both the number of data packets transmitted with and without MDDA, along with the accuracy of the sensed data. In this experiment, we are comparing the use of MDDA to not using it; therefore the device not using data suppression is treated as the "ground truth" for determining the accuracy of MDDA. The frequency of data acquisition is fixed for both groups at 10 seconds. For the MDDA group, the suppressed data is predicted to compare with the sensed data from the comparison group. For a specific time, the predicted value for a suppressed sensor is either the average of the neighboring readings (in the case of the spatial model) or a running average of the previous readings (in the case of the temporal model).

5. RESULTS AND DISCUSSION

Question 1. We evaluated how BraceForce eases the development of CPS applications for each of our three applications, we evaluated the benefits of using BraceForce specifically in terms of reduction in development time. The results show the development times for each

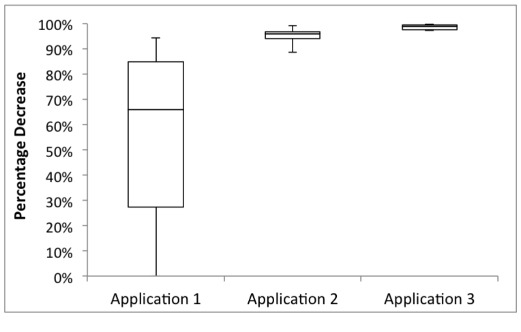

Figure 7: Decrease in development effort using BraceForce vs. Android sensor framework

of three evaluation applications when using BraceForce are not substantially different, even though the applications are increasingly complex. However, the development times when using the Android SDK increase dramatically among evaluation applications (a 349.8% increase from Application 1 to Application 2, and then another 147.9% increase from Application 2 to Application 3). Fig. 7 shows a box-and-whisker plot of the percentage decrease in development time for BraceForce versus Android. BraceForce reduces the median development time in the range of 66% (for Application 1) to 98.8% (for Application 3) compared to the Android SDK.[5] The variance for the reduction for Application 1 is wide because several of the test subjects were able to complete the (simple) Android task very quickly. As the application complexity increases, however, BraceForce's abstractions provide significant support for the development task. All participants were able to deliver working applications in all three cases when using BraceForce, but when using the Android SDK, only half of the participants delivered a working Application 1, one participant was able to successfully implement Application 2, and no participants completed Application 3.

One of the most useful sets of feedback from the post-questionnaire explained the users' stumbling blocks with respect to completing the tasks using the Android SDK:

- "Android interface is harder to understand"
- "Complex Android APIs"
- "Android permission is complicated; hard to locate the right documents"
- "Android SDK is complex and not easier to understand, not user friendly"
- "Complex logic regarding permission, broadcast receivers and so on; distributed programming in Android is very difficult"

We also asked participants whether they agree that using BraceForce was much easier and quicker than the Android SDK and if so, why. Eleven participants responded that they "Strongly Agree" with the statement, while the remaining participant responded "Agree". As for why, the users' justifications included the following:

- "Easy to manage and read, simplify the implementation"
- "Much more efficient, easier to use, easier to understand"
- "More convenient, more integrated in communication"
- "Convenient, cleaner, user friendly"
- "Very simple, easy to understand, elegant"
- "Very easy to use, hides all the low level details"
- "It hides the complex logic, easier to use, simple to debug"

[5]If a participant was not able to complete a specific task within the time limit (i.e., 4 hours in total), we asked the user to estimate the time he or she thought would be required to complete the task; this value was used in computing the relative development efforts.

Question 2. We evaluated the ramifications of using BraceForce to interface with the sensors in terms of the accuracy of the sensing task and the running time. Fig. 8 shows the results. The accuracy of the versions of all three applications was not noticeably different (it was marginally better using BraceForce), but Fig. 8 shows that executing through the BraceForce tiered architecture has some impact on the application's execution efficiency (in terms of how long it took the application to run, including interacting with the sensors). While this burden was relatively higher (an 8% increase) for Application 1, it was substantially lower for the more sophisticated Applications 2 and 3, since the interaction with the BraceForce architecture is amortized over more extensive and potentially distributed interactions.

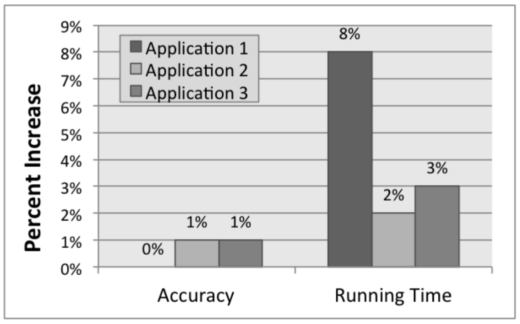

Figure 8: BraceForce's percent increase in accuracy and running time vs. Android alone

Question 3. In this experiment, we report results from using MDDA for a run of our smarthome application over three hours. We report results that examine the impact of MDDA at the aggregation level. That is, in the experiments we report in this section, we fix the frequency of data acquisition in the data layer and instead explicitly suppress updates from neighbors where the sensor readings differ by less than x% from the average of the neighboring readings. We vary x from 0.5% (the most sensitive) to 5% (the least sensitive). We report results for three sensitivity settings in Fig. 9, which plots the tradeoff of MDDA with respect to accuracy and communication costs. While MDDA at the data layer is also useful to sensing driven CPS applications, for space considerations, we focus on the higher level MDDA because it has a more significant potential to impact communication costs since the reasoning at the aggregation layer requires network communication among distributed sensing devices.

As shown, there is a modest loss of accuracy when sensor readings are suppressed, but this comes at a substantial reduction in the communication overhead. We measure both the number of packets transmitted and

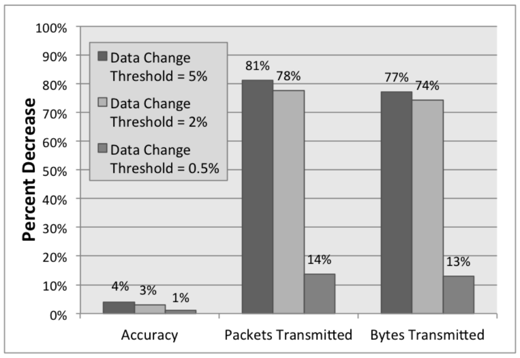

Figure 9: Tradeoffs of MDDA

the number of bytes transmitted because some communication is amortized over the entire run of the application. In our particular scenario, suppressing sensor readings that are within 2% of neighboring readings entails a only 3% loss of accuracy (in comparison to the out-of-band measured ground truth) but incurs 74-78% less communication overhead. These results are dependent on the particular model used and the ability of the phenomenon sensed to be predictably modeled. The results do show that, in applications where this is the case, MDDA whose sensitivity is tuned by the application domain expert can provide a significant benefit in terms of the cost to deploy or debug a CPS application that must acquire data from a networked set of sensors.

Discussion. We briefly discuss threats to validity.

Construct and Internal Validity. We have used two simple models for MDDA. We posit that using more sophisticated models will only bring more benefits to the suppression of superfluous transmissions of sensor data, but we have not validated this hypothesis. The application used to answer Question 3 was developed by the authors; since the focus of the evaluation for Question 3 is not on the usability of BraceForce but instead on the importance of MDDA (from a performance aspect), the particular developer should be irrelevant.

We use automated measures of CPU usage, WiFi usage, and communication latency to assess BraceForce. These statistics can be influenced by network noise, background threads, and other processes running on the test devices. To mitigate these concerns, we perform multiple trials and average values across the trials.

We use qualitative analysis of BraceForce simplicity based on Software Engineering and Computer Science students in a public university. The results may be different with professional CPS developers, however, our target is to enable *novice* programmers to build CPS applications, and our users represent these novice programmers. In our study's pre-questionnaire, the majority of our participants labeled their programming capabilities as "Average" (while five labeled their experience as "Above Average"), and only one study participant had any prior experience in interfacing with sensing devices. Future studies will include both more experienced CPS developers and even more novice programmers.

We minimize learning effects by randomizing the order of the use of BraceForce and the Android Platform. For those participants who cannot complete the evaluation applications, we use the users' estimate on how long it would take them to finish the application. It is a guess at best and it might be different (e.g., more accurate) for professional CPS developers. But we find out in the study, for those participants whose programming capabilities as "Above Average", they tend to give much higher time estimation(e.g., five times more). From this, we have a hypothesis that more experience a developer has, more accurate the time estimation is; and those with average programming skills tend to give optimistic estimation. The hypothesis is not validated though.

External Validity. We have implemented the BraceForce prototype only for the Android operating system. We therefore cannot draw concrete conclusions about the external validity in terms of BraceForce's applicability to other operating systems and platforms. We have attempted to mitigate these concerns by avoiding proprietary interfaces whenever possible and, when not possible, wrapping those interfaces in a generic way that should be repeatable for other platforms and systems. The same is true for our use of external sensors connected via the Arduino board. While we have focused our prototype implementation on the Android and Arduino platforms, we have looked at a wide variety of devices, and our design and implementation has focused on abstractions that are, in theory, easily transferable to other domains. Future work will include this translation to additional platforms as well as a novel MDDA tailored for the needs of the CPS applications.

6. CONCLUSION AND FUTURE WORK

We motivated the need for a generic and principled framework that allows for cross-platform integration of networked sensing capabilities with CPS applications. This is important in a variety of use cases, most notably to support developers of CPS applications that interact with on-board, external, and networked sensors and to support debugging of CPS applications that require direct interaction with the physical world. BraceForce provides a layered architecture that explicitly separates the application developer from the low-level complexity of interacting with sensing devices and enables the programming task to instead focus on the application logic and the integration of sensing into that logic. Our evaluation demonstrated that BraceForce does indeed lower the barrier to creating these applications. Further, BraceForce also provides two essential points in the layered architecture to perform model-driven suppression of the sensing tasks, resulting in a significant reduction in the cost of sensing (as measured in energy cost and communication cost) while mitigating the concomitant loss of sensing accuracy.

Acknowledgements

The authors thank Dr. Chien-Liang Fok and Dr. Miryung Kim for their contributions to this work. This work was supported in part by the NSF under grant CNS-1239498. Any findings, conclusions, or recomendations are those of the authors and do not necessarily reflect the views of the sponsoring parties.

7. REFERENCES

[1] Android sensor framework. `http://developer.android.com/guide/topics/sensors/sensors_overview.html`.

[2] ASM. `http://asm.ow2.org/`.

[3] F. Baccelli, N. Khude, R. Laroia, J. Li, T. Richardson, S. Shakkottai, S. Tavildar, and X. Wu. On the design of device-to-device autonomous discovery. In *Proc. of COMSNETS*, pages 1–9, 2012.

[4] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proc. of MobiSys*, 2003.

[5] 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 *Proc. of MobiSys*, pages 351–364, 2012.

[6] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[7] Sony dynamic android sensor hal. `https://github.com/sonyxperiadev/DASH`.

[8] P. Edara, A. Limaye, and K. Ramamritham. Asynchronous in-network prediction: Efficient aggregation in sensor networks. *ACM Trans. on Sensor Nets.*, 4(4), 2008.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, pages 1–11, 2003.

[10] H. Gupta, V. Navda, S. Das, and V. Chowdhary. Efficient gathering of correlated data in sensor networks. *ACM Trans. on Sensor Networks*, 4(1), 2008.

[11] W.B. Heinzelman, A.L. Murphy, H.S. Carvalho, and M.A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[12] J. Hill and D. Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, UC Berkeley, 2002.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, 2000.

[14] V. Jakkula and D.J. Cook. Mining sensor data in smart environment for temporal activity prediction. In *Proc. of KDD (Poster)*, 2007.

[15] jsonbeans. `http://code.google.com/p/jsonbeans/`.

[16] E. Kuleshov. Using ASM framework to implement common bytecode transformation patterns. In *Proc. of AOSD*, 2007.

[17] P. Levis. Experiences from a decade of TinyOS development. In *Proc. of OSDI*, 2012.

[18] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of ASPLOS*, pages 85–95, 2002.

[19] F.X. Lin, A. Rahmati, and L. Zhong. Dandelion: A framework for transparently programming phone-centered wireless body sensor applications for health. In *Proc. of Wireless Health*, 2010.

[20] C. Liu, K. Wu, and J. Pei. An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. *IEEE Trans. on Parallel and Distributed Systems*, 18(7):1010–1023, 2007.

[21] S. Nadimi and B. Bhanu. Physics-based models of color and ir video for sensor fusion. In *Proc. of MFI*, pages 161–166, 2003.

[22] S. Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *Proc. of MobiSys*, pages 29–42, 2012.

[23] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A.Y. Ng. ROS: An open-source robot operating system. In *Proc. of the Open Source Software Workshop of ICRA*, 2009.

[24] U. Raza, A. Camerra, A.L. Murphy, T. Palpanas, and G.P. Picco. What does model-driven data acquisition really achieve in wireless sensor networks? In *Proc. of PerCom*, pages 85–94, 2012.

[25] T. Schreiber. Android binder: Android interprocess communication. Master's thesis, Ruhr-Universitat Bochum, 2011.

[26] C. Yang and R. Cardell-Oliver. An efficient approach using domain knowledge for evaluating aggregate queries in WSN. In *Proc. of ISSNIP*, 2009.

[27] C. Yang, R. Cardell-Oliver, and C. McDonald. Combining temporal and spatial data suppression for accuracy and efficiency. In *Proc. of ISSNIP*, 2011.