Building a Practical Sensing System

Robert S. Moore*, Bernhard Firner†, Chenren Xu†, Richard Howard†, Yanyong Zhang†, Richard P. Martin*
*{romoore, rmartin}@cs.rutgers.edu, †{bfirner, lendlice, reh, yyzhang}@winlab.rutgers.edu
*Department of Computer Science, Rutgers University, Piscataway, NJ, USA
†WINLAB, Rutgers University, North Brunswick, NJ, USA

***Abstract*—Developing Internet of Things (IoT) applications can be a complex task for many developers, requiring knowledge of sensor hardware, deployment characteristics, network limitations, and multiple protocols. Because of this, IoT development has been largely centered around research scientists and domain experts, with only a limited number of simple applications coming from the broader community. We present a software architecture that seeks to simplify and accelerate the development of IoT applications, making them accessible to a larger community of developers. To keep the system simple yet flexible, it focuses on a limited number of abstractions, relying on the developers and administrators to enforce additional constraints where necessary. We present a model system and a prototype implementation, along with experiences developing applications.**

I. Introduction

The Internet of Things may prove to be a powerful and disruptive conceptthat can improve the way people live by creating new ways to perform environmental monitoring, home automation, power conservation, health care, and work flow optimization. For most developers, however, these systems are by and large inaccessible, due in large part to the difficulty of deploying and managing these systems. This is a serious bottleneck for the adoption of the Internet of Things. In this work we address complexity of use and development in IoT systems. The goal of this research is to make application development accessible to the vast majority of developers.

We begin by describing the design and deployment of a new software architecture, *Owl Platform* that eases the burden of developing IoT applications. Owl Platform allows someone with modest programming knowledge (1-2 years of programming coursework) and no previous experience with sensor networks the ability to design and deploy meaningful IoT applications. We qualitatively demonstrate our claims by experimenting with a dynamically growing sensor deployment in an office environment with offices, eating, meeting, and storage areas.

A primary conclusion of our experiences is that a clear separation of concerns between the roles of users, application developers, sensor designers, and system administrators reduces the complexity of application development and sensor deployment. Separating these concerns allowed novice programmers to develop applications with only intermediate levels of knowledge in a single area of the deployment. This separation of concerns is supported through two abstractions layers, pictured in Figure 1.

The first abstraction layer, called the *world model*, is a virtual representation of a physical space, the items in that space, and those items' attributes. Items may be physical (a chair) or abstract (a meeting) and there are no constraints or assumptions made about items or which attributes they possess. This stands in contrast to location-based systems such as BAT [7] and SenseWeb [12] which focus on location-based services and queries. Instead, the Owl Platform world model focuses on implicit relationships based upon a hierarchical name space (e.g. the names of all lights in a building begin with <building>.light) or explicit relationships based upon time-varying item attributes. This approach is similar in spirit to LDAP [10] or SNMP servers [16] and provides the flexibility to store and distribute abstract information, rather than just raw sensor data.

The second layer of abstraction is the Owl Platform *aggregator*. The aggregator hides the low-level details of individual sensors from the application developer while also hiding the high-level details of a sensor's use from a person deploying sensors. This approach allows sensor designers and application developers to work separately by giving them both a simple standard to work with. For example, a developer who designs and deploys simple switch and temperature sensors only needs to worry about communication with the aggregator. Likewise, someone with application knowledge can use these sensors without sensor-specific knowledge; combining the application code in the sensor would only complicate the design and slow the process of building sensors.

Owl Platform advances the state of the art in IoT systems by providing simple, flexible interfaces that enable rapid development, deployment, and management of novel applications. We also demonstrate that it supports a wide range of sensing-based applications and fosters creative and novel applications by making these systems available to a larger community of developers.

In this paper we survey related work in middleware and service-oriented systems in Section II, and discuss the additional features required to create multi-user systems that do not require specialized domain knowledge in Section III. In Section IV, we present our proposed system, detailing how the world model and aggregator abstractions work to simplify interaction with the system without sacrificing flexibility. We evaluate the performance of our system during a year-long deployment supporting smart office applications in Section V.

II. Related Work

Modern middleware systems are designed to aggregate data from many sensors and sensor networks into a common interface, often in the form of a software Application Programming Interface (API). Systems like SenseWeb [12] and Global Sensor Network (GSN) [3] are effective at providing a neutral platform for many different sensors. Unfortunately

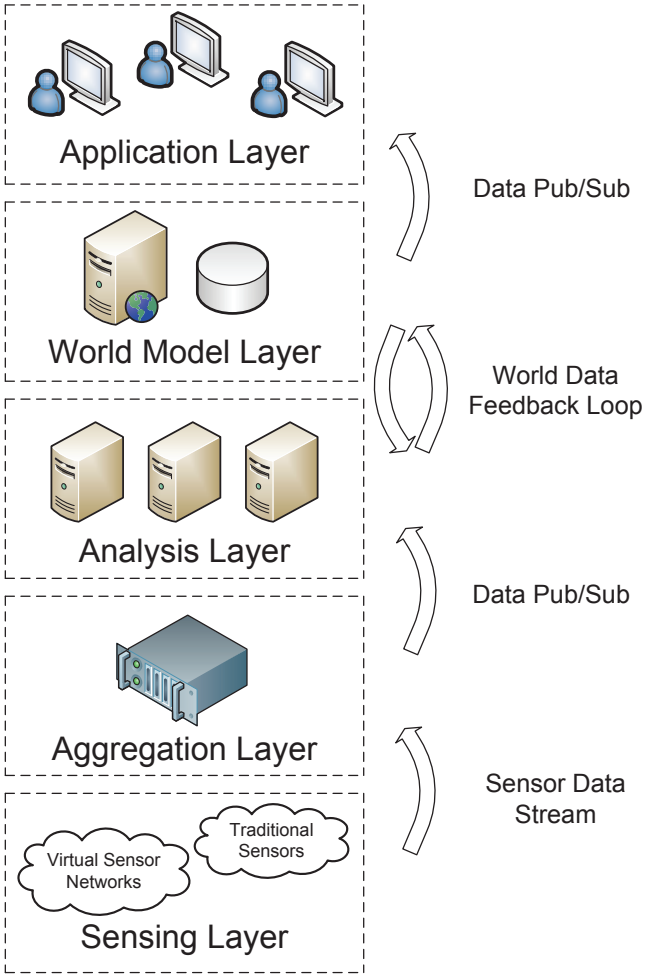

Fig. 1: Two layers of abstraction, a world model and an aggregator, separate concerns across sensor deployments, information processing, and application development.

their focus on the physical sensors and data mean that they are more appropriate for domain experts than novice users.

The BAT system [7] provides a good example of how a system can be well-designed for its initial purpose, but then fails to adapt to the demands of a changing user base. When the BAT system was redeployed at a second location, Mansley et al. [14] found that its initial set of features did not satisfy the diverse requirements of its new user base. Similarly, Microsoft's SixthSense [15] platform created a limited set of information through its "inference engine" and allowed for powerful SQL queries on that data, but was limited by the expressiveness of SQL. An effective middleware system must combine useful data abstractions with an API that allows users to analyze the data flexibly, without the system becoming so complex that only experts are capable of using it.

By designing the system so that non-experts can use it, a community can form around the platform. An effective example of this is the Firefox web browser, which provides a simple but powerful add-on API [1] that enables users to expand the functionality of the web browser. Some research projects, like Common Sense [17], have involved the user community to improve the system. In this case the researchers distributed air quality sensors to users who carried them during their daily activities. The users were able to gather, annotate, and discuss the data, but were not able to interact more fully with the data or modify the system to suit their own needs.

Creating a user base or community around a sensor network platform opens up many new possibilities. For the Internet of Things, providing a flexible and simple API enables users to add new functionality to the system, adapt it to their own needs, or even provide additional interfaces to the system. Separating the data gathering platform from the analysis platform means that multiple sensor networks can coexist without interference. Additionally, users can keep their own analysis systems running separately from the "official" system run by researchers or sponsors. This avoids data contamination for researchers, since their systems are unchanged, but allows the system to grow to suit the needs of new communities of users. This in turn can provide the same researchers with powerful new tools or ideas that they can use to improve the system.

III. System Requirements

Al-Jaroodi and Mohamed [4] performed a survey of multiple service-oriented middleware systems and identified a set of nine requirements considered important. These requirements are: creating, publishing, and discovering information, supporting heterogeneous systems, integration transparency, adaptation, scalability/efficiency, reliability/security, and quality of service.

1) Creating, Publishing, and Discovering Information: The first three requirements address the basic features to make data available to users. First, the middleware system needs to provide a common API for developers to create new data analysis software which works across many different platforms. This enables seamless integration of multiple sensor networks, preventing the system from being bound to specific hardware or software environments. Second, there must be a way to share this new software with other users via registration or publication services in the system. Third, there must be a way for users to discover and use these new services. Existing systems have addressed these issues well, usually having well-defined semantics for producing, consuming, combining, and publishing data from various sensor sources [5].

2) Heterogeneity, Integration Transparency, and Adaption: The next three requirements govern hiding a system's complexity from users. APIs and protocols should be available across platforms, the details of services should be integrated into the system so that their complexity is hidden from users, and the system should adapt to component failures as seamlessly as possible without user intervention. These are arguably the most important as they determine how much effort is required of developers and users to work with and expand the capabilities of a system,

As mentioned in Section II, separating the sensor network abstraction layer from the data analysis layer allows the system to be more flexible without adding unnecessary complexity. A common approach to designing a cross-platform system is to provide access through a network API. This is an effective way

to allow the system to adapt to user needs and prevents it from being bound to a specific hardware or software environment.

3) Scalability, Reliability, Quality of Service: The idea of scalability and efficiency in a middleware system applies to difficulties in processing, storing, and disseminating the potentially large amounts of data handled. Most data queries will not involve the entire system, but will most likely be limited to a specific spatial, temporal, or relational domain. Knowing whether it is raining at the user's office is a more common query than knowing how much rainfall occurs on average across the entire globe. Though both queries are possible in such a system, the former is more likely and should be a more important consideration when designing the system.

Security in such a system is another important consideration since many users will be accessing it simultaneously. By identifying each piece of information in the system with an "origin" identifier, users can be sure of the authorship of data. The user can then provide a set of origin preferences to the system, which can manage the flow of information as different sources are available. While this is an important part of such a system, a thorough discussion of security is outside the scope of this paper.

Finally, quality of service concerns vary with the type of data being provided and the application being designed. Users might expect to have regular updates from many types of sensors (temperature, humidity, light, etc.), while other types of data may only be sent "on-demand" (events, query results). In any case, it is reasonable to expect the user interface to provide further quality of service information such as failure notifications, heartbeats, and ensuring that periodic data arrives on-time and with as little irregularity as possible.

IV. Owl Platform Architecture

Our proposed IoT middleware system, called *Owl Platform* has several key features that allow it to support many users of differing skill levels. Owl Platform achieves effective component segregation and simple interactions by promoting two primary abstraction layers to separate concerns into three "views" of the entire application stack, pictured in Figure 2. Each of these layers supports an API over TCP/IP.

We will describe the application developer's view and the data analyst's view of the system below. We will address the sensor expert's view in future work.

A. The Application Developer's View

The application developer views the system as just the world model, a named hierarchy of physical and conceptual objects and their attributes, similar in spirit to LDAP or SNMP. Each item in the world model is a name and a set of attributes with primitive or user-defined types. In Owl Platform, every attribute has a creation time and an expiration time that denote when the value is valid. Figure 2 shows a view of the world model. Users query the world model by searching for items with names matching a given POSIX RegEx [2], a set of desired item attributes, and a time range for the query. Queries can also request subscriptions for data as it arrives rather than data in an historic time range.

Names do not change over time so only immobile items should have implicit locations in their names. Attributes can change over time, so if an object moves it would have its location explicitly stated as an attribute rather than implicitly stated in its name. Thus a client that wishes to draw the current locations of all mobile items would request all items (using the ".*" RegEx) that had a "location" attribute.

1) Attribute Names and Data Types: An attribute name strictly specifies the data type of the attribute, similar to MIME types. For instance, "temperature.Celsius" might specify "a 64-bit IEEE floating point value representing the temperature in degrees Celsius." In addition to simple types, the system can also recognize aggregate types, such as vectors of primitive types. Primitive data types are specified in a standards document and libraries in multiple languages can be provided to interpret these data types. This allows the system to support any arbitrary type while still having a standard that specifies how to interpret or display each kind of data.

2) Attribute Origins: The world model remembers the origin of each attribute. This origin specifies the source that provided the attribute to the world model and, along with a digital signature, can be used to provide authenticity for world model data. The origin string also allows clients to differentiate different sources of similar data. Two different sources could provide the same information to the world model, for instance by using a different algorithm or hardware to generate the same data. The attribute origin field gives clients and solvers that interact with the world model a way to specify a preference for one source of data over another. This provides a method for fault tolerance - if the preferred source of data fails, because of sensor failure, a software fault, or any other reason, then the world model can immediately serve the most preferred data from the remaining alternatives.

B. The Data Analyst's View

The data analyst views the system as an aggregator with raw sensor data and a world model with processed sensor data and other high-level information. Analysis software, called *solvers* in Owl Platform, subscribe to sensor data from the aggregator by specifying patterns of physical layer IDs and sensor IDs. Solvers request data from the world model in the same way that client applications query the system. Solvers are also free to use information from sources outside of the aggregator and world model. When solvers create new data they send it back to the world model. This mechanism allows each solver to be a standalone process independent of other components.

The world model and aggregator APIs allow developers to use any combination of platforms and development tools that support TCP/IP, from smart phones to desktop computers. Although the possible complexity of a solver is high, we advocate using the UNIX philosophy of small, independent units whose results can be combined to create high-level results, encouraging data reuse.

The simplest solvers take raw sensor information from the aggregator, process it, and put it into context in the world model. For example, a temperature value from a sensor will be processed by a temperature solver and associated with an item in the world model. More complicated processing of that

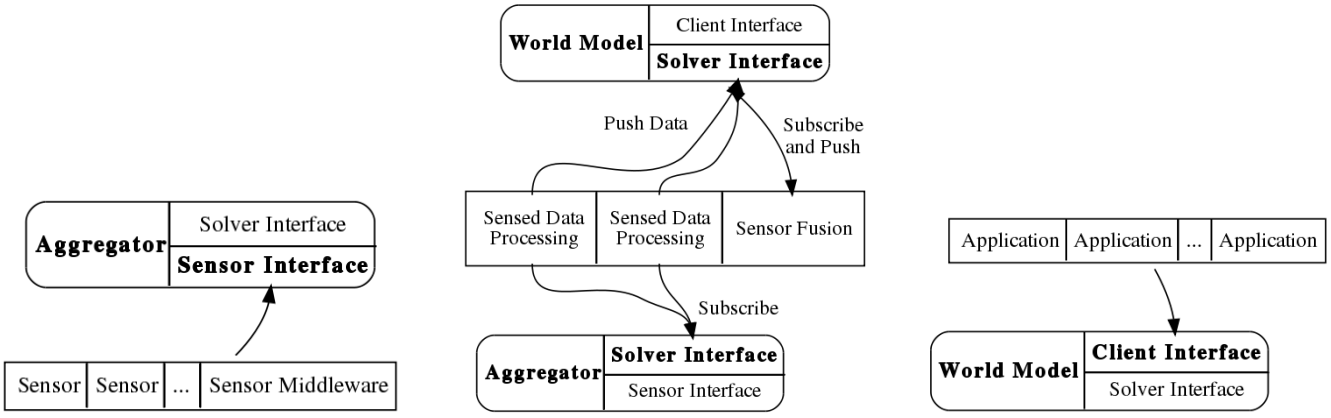

Fig. 2: The world model and aggregator partition the development space into separate views for deploying new sensors (a), analyzing sensor data or performing higher level analysis (b), and developing applications using the information in the system (c).

data is left to other solvers. This means that many solvers do not process sensor data directly and will never need to connect to the aggregator.

1) Transient Solvers: The solvers we have described do not need to interact with clients or solver requests and can be relatively simple because requests stop at the world model and do not go to the solver itself. However, this might not be suitable for all situations as the domain of some kinds of data is too large to fully compute and add into the world model. This includes, at the extreme, the space of all possible logical queries. A slightly more modest example is a solver that computes the covariance of the signal strength values in the system.

As the number of transmitters increases, the number of results that a covariance solver would need to compute and place into the world model would increase factorially. To avoid this kind of situation the world model API allows solvers to declare that they have *transient* data. The world model keeps a list of attributes that are provided by transient solvers. When the world model receives a request for that a transient data type, the world model forwards the request unchanged to the appropriate solver. The transient solver then generates that information "on demand." This allows the system to support solvers of arbitrary complexity.

V. Core Owl Platform Deployment

The system was initially deployed with a small number of sensors and applications, and was grown over the course of a year. Many sensors and solvers were added to the system "live" during the deployment, and in this section we will focus on a few illustrative examples. In Section VI we will discuss how users and new developers used and expanded the system for summer projects, demos, research, and fun.

A. Initial Deployment

The deployment covers an area of approximately 66.5 by 51.5 meters (3,425 m^2) and contains cubicles, conference rooms, a kitchen, storage room, and lab areas.

1) Radio Transmitters: Our physical layer consists of pipsqueak radio tags [6]. The pipsqueak broadcasts its unique sensor ID every second. The beacons' signal strengths are used by a set of core Owl Platform solvers: localization [8], location discrimination, and mobility detection [9]. These tags were chosen because of their size and lifetime - they are approximately 3.5cm by 3cm and this version runs for over 2 years on a coin cell battery at this duty cycle.

2) Signal Strength Based Solvers: The *signal strength statistics solver* uses information from the aggregation layer to calculate the mean, median, and variance of wireless link signal strengths and the average signal variance of each transmitter's signals across all links. To avoid unnecessary data creation and transmission we wanted this data to be created "on demand" so we made this a transient solver, as described in Section IV-B1.

The *mobility detection solver* detects mobility events using changes in signal variance from a transmitter at multiple receivers. With multiple deployed receivers this technique gives low false positives and high detection rates (close to 99%), even for mobility events lasting only a few seconds [9]. This solver requests signal variance measurements from the world model that were originally created by the signal strength statistics solver. The solver updates the mobility attributes of items in the world model.

The *localization solver* is written based on a Bayesian localization algorithm [13], which uses signal strength measurements. The solver uses real-time training data from transmitters pinned to the wall at known locations, similar to the approach in the LEASE system [11]. The locations of immobile transmitters and receivers are stored in the world model. The localization solver subscribes to mobility information from the world model to trigger localization and median signal strength values to perform localization.

As we added transmitters and receivers in new locations the distributed nature of the system allowed us to easily overcome networking issues, such as administrative boundaries and data collection across distant physical locations. Some of the gateway hubs were split across two different networks, each

separated by a Network Address Translation (NAT) firewall that blocks many connection types. The solution to this was relatively simple; we ran two aggregators, one for each wired network domain, and solvers connected to both aggregators to obtain information across both networks. Later we also expanded the Owl Platform deployment to a different building 8 miles away. We used tiered aggregators to keep information across multiple sites in a centralized location, with a solver forwarding information from one aggregator into the other.

B. Adding Sensing

We also added a variety of sensors to provide critical information for the experimental deployment, including temperature, chair use, door states (open or closed), and power consumption.

We used these different sensors to build multiple solvers: checking for propped open doors, detecting room use and social gatherings, fresh coffee brews, and so on. We found it very easy to deploy new sensors and solvers - generally the most difficult task was to properly package sensors for long-term use. The abstraction layers allowed us to focus on these physical problems though, so we did not need to worry about new analysis software or networking issues. We will describe our experiences sensing fresh coffee brews to illustrate the little effort needed to work with Owl Platform.

The microcontroller in the tags has an on-chip temperature sensor which we used to detect coffee brews. Initially we deployed a sensor inside the upper chamber of the coffee pot to sample heat from steam as soon as coffee was brewed. Once the sensor was deployed, we added an item into the world model named <region>.coffee pot.kitchen to indicate that this item was a coffee pot for the kitchen. We added an attribute to this item called "sensor.temperature" with the ID of the temperature sensor. We wrote a temperature solver that searches the world model for temperature sensors, requests those sensors' data from the aggregator, and adds "temperature.Celcius" attributes to items with temperature sensors. We then wrote a second solver to search for all items with "coffee pot" in their names that had temperature attributes. When this solver observes a local temperature maximum it updates the "fresh coffee" attribute of the coffee pot to indicate the last time that coffee was brewed.

The location where we placed the temperature sensor had a high failure rate however, because the high temperature and humidity melted, warped, or corroded various packaging attempts. After several different configurations we eventually encased our sensor in shrink wrap, placed it inside a rubber balloon, and taped it to the outside of the machine near the heated water reservoir. Even though we changed sensors several times, the Owl Platform system made this very painless - we simply changed the "sensor.temperature" attribute of the coffee pot to indicate the current sensor. No software was changed.

Eventually, the original coffee pot was replaced with a different model with two independently heated plates and a permanently heated water reservoir. This makes results from a temperature sensor more ambiguous and results from a power sensor complicated to interpret because there are three independent elements that draw power. Eventually we used a switch sensor to detect when people opened the lid to the water reservoir to fill it. This kind of change does require a change in the coffee solver - however, this change still leaves all client software unaffected because Owl Platform has hidden the details of that decision from client applications. When the coffee solver updates the "fresh coffee" attribute of the coffee pot client applications do not know how the solver creates its solution.

VI. Results: User and Developer Experiences

We must evaluate the effectiveness of Owl Platform for two groups of people - users and developers. Users simply want to retrieve information from the system through graphical tools and event notification systems. They will only interact with the client interface of the world model. Developers want to use the system to build something new, so they might deploy new sensors, write new software, create new applications, or any combination of the three.

Information created by solvers is stored in the world model so users rely upon clients to fetch that information for them. Initially we built a live status map of the system to track the locations and status of items in the deployment area. Icons on the map changed to indicate status updates. Door icons open and close along with their real-life counterparts, chairs show a person sitting when they are being used, projectors "light up" when their power is on, and so on. This status map makes a good demo and is suitable for status lookups, such as discovering room use information and locating lost items. This is particularly helpful in cases when the system cannot predict when the user may need the information. For example, a user will look at the status map when he/she needs a conference room.

In addition to the live status map, we also provide a "push" mechanism which sends event updates to users over SMS, email, and Twitter. For example, although information on coffee brew times was available on the status map by moving a mouse cursor over the coffee machine icon, if the user is always ready for a cup of fresh coffee whenever possible, it is much more desirable for Owl Platform to push the event to the user. Similarly, when tea time was detected by a solver (through a gathering of mugs in the kitchen) email notification was preferred to checking on the status map.

As a final example, we implemented a device-free passive localization and counting solver that determines the number and location of people based upon their impact on ambient radio signals [18], [19]. Passive localization can trace people without revealing their identity, and is suitable for applications with strong privacy concerns. In its current state, we can only localize a small number of people (4) in the same room. We are exploring the possibility of integrating a camera or off-the-shelf smartphones to bring more sensor modalities to extend our capabilities and track people [20].

VII. Conclusions and Future Work

We addressed the complexity of multi-user pervasive systems and introduced a new architecture, named *Owl Platform*, designed to lower the barriers to entry for novice users. To demonstrate our solution we described an experimental multi-user deployment that expanded over time. The system's

abstractions and APIs successfully allowed sensor developers, software analysts, and application developers, and information consumers to simultaneously use the system without interfering with each other. New hardware and software was added to the deployment "live" without disrupting any running services.

Owl Platform focuses on supporting hardware and software heterogeneity through network APIs, hiding system complexity through the world model and aggregator abstraction layers, and adapting to dynamic deployments. This makes Owl Platform suitable for users with different knowledge levels and goals. In contrast, traditional sensor network middleware systems focus on data creation and publishing for expert users, and are often limited to a specific application domain.

There are three areas of discussion that we have left for future work: data authenticity, security and privacy, and large-scale deployments. In Section III we proposed digital signatures as a method to assure data authenticity. We also note that access controls similar to those found in file systems may be well-matched to the hierarchical naming structure used in the world model. These access controls could be a good method for providing security and privacy within the system.

In addition we recognize multiple open questions with respect to large-scale deployments. When they cross multiple political and administrative boundaries, should each system remain separate or are distributed systems more appropriate? When handling client queries that require remote information, is redirection the correct approach, or would a peering network with access control be better? We are currently exploring these problems by deploying Owl Platform systems in more locations, and in future work we hope to be able to present interesting solutions in this area.

REFERENCES

[1] Add-ons for Firefox. Website, Retrieved April 6, 2011. https://addons.mozilla.org/en-US/firefox/.

[2] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). System Interfaces. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6.*, 2004.

[3] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *IEEE MDM*, 2007.

[4] J. Al-Jaroodi and N. Mohamed. Service-oriented middleware: A survey. *Journal of Network and Computer Applications*, 2012.

[5] J. Anke, J. Müller, P. SpieSS, L. Weiss, and F. Chaves. A service-oriented middleware for integration and management of heterogeneous smart items environments. In *MiNEMA*, 2006.

[6] B. Firner, P. Jadhav, Y. Zhang, R. Howard, W. Trappe, and E. Fenson. Towards Continuous Asset Tracking: Low-Power Communication and Fail-Safe Presence Assurance. In *IEEE SECON*, 2009.

[7] R. K. Harle and A. Hopper. Deploying and Evaluating a Location-Aware System. In *ACM MobiSys*, 2005.

[8] K. Kleisouris, Y. Chen, J. Yang, and R. Martin. The Impact of Using Multiple Antennas on Wireless Localization. In *IEEE SECON*, 2008.

[9] K. Kleisouris, B. Firner, R. Howard, Y. Zhang, and R. P. Martin. Detecting Intra-Room Mobility with Signal Strength Descriptors. In *ACM MobiHoc*, 2010.

[10] V. Koutsonikola and A. Vakali. Ldap: framework, practices, and trends. *IEEE Internet Computing*, 2004.

[11] P. Krishnan, A. S. Krishnakumar, W.-H. Ju, C. Mallows, and S. Ganu. A System for LEASE: Location Estimation Assisted by Stationary Emitters for Indoor RF Wireless Networks. In *IEEE INFOCOM*, 2004.

[12] L. Luo, A. Kansal, S. Nath, and F. Zhao. Sharing and exploring sensor streams over geocentric interfaces. In *ACM GIS*, 2008.

[13] D. Madigan, E. Einahrawy, R. Martin, W.-H. Ju, P. Krishnan, and A. Krishnakumar. Bayesian Indoor Positioning Systems. In *IEEE INFOCOM*, 2005.

[14] K. Mansley, A. R. Beresford, and D. Scott. The carrot approach: Encouraging use of location systems. In *ACM UbiComp*, 2004.

[15] L. Ravindranath, V. N. Padmanabhan, and P. Agrawal. SixthSense: RFID-based Enterprise Intelligence. In *ACM MobiSys*, 2008.

[16] W. Stallings. Snmpv3: A security enhancement for snmp. *IEEE Communications Surveys Tutorials*, 1998.

[17] W. Willett, P. Aoki, N. Kumar, S. Subramanian, and A. Woodruff. Common sense community: Scaffolding mobile sensing and analysis for novice users. In *Pervasive*. 2010.

[18] C. Xu, B. Firner, R. S. Moore, Y. Zhang, W. Trappe, R. Howard, F. Zhang, and N. An. Scpl: indoor device-free multi-subject counting and localization using radio signal strength. In *ACM/IEEE IPSN*, 2013.

[19] C. Xu, B. Firner, Y. Zhang, R. Howard, J. Li, and X. Lin. Improving rf-based device-free passive localization in cluttered indoor environments through probabilistic classification methods. In *ACM/IEEE IPSN*, 2012.

[20] C. Xu, M. Gao, B. Firner, Y. Zhang, R. Howard, and J. Li. Towards robust device-free passive localization through automatic camera-assisted recalibration. In *ACM SenSys*, 2012.