Pervasive Computing for the 99%

Bernhard Firner
WINLAB, Rutgers University
671 Route 1 South
North Brunswick, N.J.
08902-3390
bfirner@winlab.rutgers.edu

Robert S. Moore
CS Dept., Rutgers University
110 Frelinghuysen Rd.
Piscataway, NJ 08854
romoore@cs.rutgers.edu

Richard Howard
WINLAB, Rutgers University
671 Route 1 South
North Brunswick, N.J.
08902-3390
reh@winlab.rutgers.edu

Richard P. Martin
CS Dept., Rutgers University
110 Frelinghuysen Rd.
Piscataway, NJ 08854
rmartin@cs.rutgers.edu

Yanyong Zhang
WINLAB, Rutgers University
671 Route 1 South
North Brunswick, N.J.
08902-3390
yyzhang@winlab.rutgers.edu

Technical Report DCS-TR-691
Department Of Computer Science
Rutgers University, New Brunswick, NJ 08901
November, 2011

ABSTRACT

A key limiting factor for the pervasive community has been the difficulty developing and deploying general purpose systems. Such systems should make application development easy, support a wide range of devices and sensors, and allow users to share these resources. Designing a multi-user middleware system that allows novice users to add arbitrary hardware and software raises several challenges, such as resolution between conflicting and stale data, managing data dependencies as software and hardware is changed, and tradeoffs between complexity and expressiveness in the API of such a system. We will discuss the feature set that could solve these problem, and test these features through a software implementation. We then evaluate the system after a year long deployment supporting smart office applications.

1. INTRODUCTION

Pervasive sensor systems and their applications are disruptive technologies that can improve the way people live by creating new ways to perform environmental monitoring, home automation, power conservation, health care, and work flow optimization. Mark Weiser's wider vision of pervasive computing [16], however, is still unrealized, due in large part to the difficulty of deploying and managing these systems. This is a serious bottleneck for the adoption of pervasive systems.

Although many middleware layers have been proposed to address this complexity by simplifying management of sensors and providing simple APIs to search and process sensor information, the target users of these systems have been domain experts. Just providing easy access to sensor data may not be enough to encourage wide-spread adoption of pervasive systems - there are still large barriers between non-experts and the knowledge hidden inside of pervasive sensing deployments.

In this work we address complexity of use and development in pervasive systems. The goal of this research is to make pervasive computing application development accessible to the vast majority of developers - the 99%. Our approach stands in contrast to other efforts, where the use of novel operating systems and programming environments makes pervasive development only accessible to the gifted - the 1%.

We begin by describing the design and deployment of a new software architecture, *Octopus*[1] that eases the burden of developing pervasive applications. Octopus 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 pervasive applications. Applications include managing the heating and cooling in a room, sending notifications when items leave an area, or playing customized music play lists depending on who is in the room.

We qualitatively prove our claims by experimenting with a dynamically growing pervasive sensor deployment in an office environment with offices, eating, meeting, and storage areas. Our approach is to have computer science and

[1]Octopus is an outgrowth of our experience with the GRAIL real-time location tracking system. It is version 3 of the design. The primary functional difference from Version 2 is that Octopus supports sensing and related application extensions in addition to tracking. We plan to add actuation to GRAIL in Version 4.

engineering undergraduates develop pervasive applications using Octopus. 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 the junior students 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.

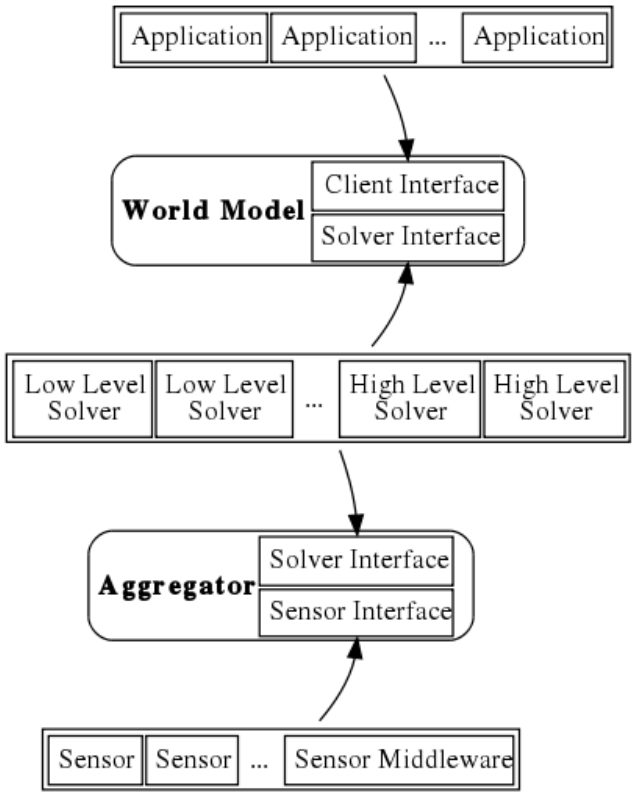
The first abstraction layer, called the *world model*, is a virtual representation of a physical space, the items in that space, and those items' attributes. Items may be physical (a chair) or abstract (a meeting) and there are no constraints or assumptions made about items or which attributes they possess. This stands in contrast to location-based systems such as BAT [8] and SenseWeb [13] which focus on location-based services and queries. Instead, the Octopus 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 [11] or SNMP servers [18] and provides the flexibility to store and distribute abstract information, rather than just raw sensor data.

The second layer of abstraction is the Octopus *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.

Further, we note that the Internet of Things has received much attention in the pervasive computing community. Although networking everyday objects will increase the applicability of pervasive computing, many of these efforts take an overly data-centric view. For example, users might not care to know the state of a light bulb remotely, but rather would like an application to manage the lighting of their home that balances energy costs, lighting needs, and ease of use.

Octopus advances the state of the art in pervasive computing by allowing sensor designers, application developers and system administrators clear roles. Our focus is on easing application development because it is impossible to anticipate every user's needs. As recent advances in web browsers and smart phones have allowed an explosion of new and creative applications for these environments, Octopus will allow new and creative uses of pervasive computing environments.

In this paper we survey related work in middleware and service-oriented systems in Section 2, and discuss the additional features required to create multi-user systems that do not require specialized domain knowledge in Section 3. In Section 4, 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 5.



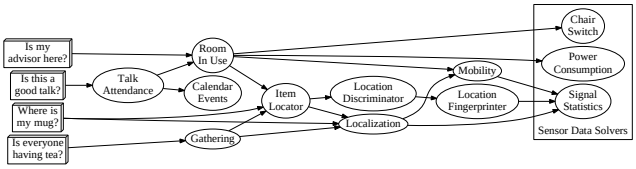
1: Two layers of abstraction, a world model and an aggregator, separate concerns across sensor deployments, information processing, and application development.

2. 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 [13] and Global Sensor Network (GSN) [4] are effective at providing a neutral platform for many different sensors. Unfortunately their focus on the physical sensors and data mean that they are more appropriate for domain experts than novice users.

The BAT system [8] 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. [15] found that its initial set of features did not satisfy the diverse requirements of its new user base. Similarly, Microsoft's SixthSense [17] 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



2: Possible questions users might ask of a smart office system and services needed to answer those questions. Arrows represent a data dependence from one unit to another.

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 [19], 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. In the case of pervasive computing, 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.

3. SYSTEM REQUIREMENTS

Al-Jaroodi and Mohamed [5] 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.

3.0.1 *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 [6].

3.0.2 *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 2, 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.

Another effective technique for hiding a system's complexities is automating data dependency management. If a user requests access to a type of information that requires other information to produce it, the system should transparently perform any necessary steps to enable access to those data sources as well. This is no small task, and careful consideration should be given to ensure that the dependency management system is both robust and transparent.

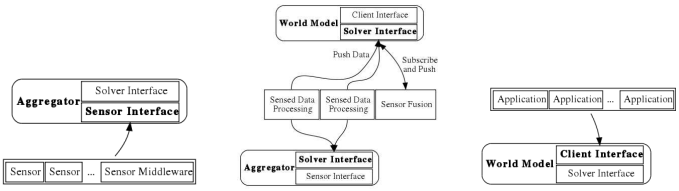
3.0.3 *Scalability, Reliability, Quality of Service*

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.

System reliability can be achieved through a combination of techniques, including distributing the components across multiple physical locations and providing redundant data sources and storage devices. These steps protect against physical failures, and can also make the system more robust by reducing the possibility of bottlenecks in data distribution. 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.

Using digital signatures to identify the origin of information not only protects against malicious or misbehaving components, but allows the system to combine multiple sources of related data (different levels of location information) into a single stream of information. 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 also 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



(a) (b) (c)

3: 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).

4: A screen shot of one of the GUI interfaces to the system. This GUI is a flash program viewable through web browsers.

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.

4. OCTOPUS ARCHITECTURE

Our solution is a pervasive computing middleware called Octopus. This system has several features that allow it to support many users of differing skill levels. A key observation is that any system must hide interactions between users across different conceptual layers (sensing, data analysis, and client applications) and with varying goals. The design challenge is to provide such a system while still keeping development simple. Octopus achieves this by using two abstraction layers to separate concerns into three "views" of the entire application stack, pictured in Figure 3. Each of these layers supports an API over TCP/IP.

First we will describe the two simpler views of the system, the application developer's view and the sensor expert's view. Then we will explain the role of the data analysis view and show how this layer bridges the gap between sensor data and end users while still providing a simple API for data analysis and storage.

4.1 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 Octopus, every attribute has a creation time and an expiration time that denote when the value is valid. Figure 3 shows a view of the world model. Users query the world model by searching for items with names matching a given POSIX RegEx [3], 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.

The hierarchical naming structure allows Octopus to capture the implicit relationships between items - for instance the URI structure <location>.<type class>.<name> implicitly captures the relationship between items in the same location or of the same type class. If a client wishes to find information for all of the doors located on the 2^{nd} floor of the Louvre it could search for objects with the RegEx "Louvre.2.door.*".

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.

4.1.1 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.

4.1.2 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.

4.2 The Sensor Expert's View

The sensor expert's view of the system is a sink node that expects data in a specified format. A sensor provides the aggregator with its physical layer, sensor ID, and sensed data. The physical layer identifies a particular kind of sensor or virtual sensor, such as GSN [4].

To store configuration information about deployed sensors, the sensor expert adds information into the world model, for instance with the GUI tool pictured in Figure 4. Storing

the configuration information in the world model means that analysis software queries the world model for that information, not the sensors. This allows the system to support low power and transmit-only sensors because they do not need to respond to requests.

This does not restrict sensors from being more interactive - if a sensor is powerful enough (*e.g.* a cell phone) it can interact with the system in the analysis layer rather than in the sensor layer. This layer is for sensors and sensor networks that are too energy- or resource-constrained to perform their own analysis or that have programming models or network architectures that make data analysis very difficult.

4.3 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 Octopus, 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 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.

4.3.1 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.

(a) (b)

5: A stationary transmitter pinned to the wall, (a), beacons from a known location. Gateway hubs with attached receivers, (b), receive the packets and forward data and received signal strength information to the aggregator.

5. CORE OCTOPUS 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 6 we will discuss how users and new developers used and expanded the system for summer projects, demos, research, and fun.

5.1 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. The user interface in Figure 6.1(a) shows the building floor plan with some of the tracked objects as icons.

5.1.1 Radio Transmitters

Our physical layer consists of pipsqueak radio tags [7], pictured in Figure 5(a). The pipsqueak broadcasts its unique sensor ID every second. The beacons' signal strengths are used by a set of core Octopus solvers: localization [9], location discrimination, and mobility detection [10]. These tags were chosen because of their size and lifetime - they are approximately 3.5cm by 3cm and this version runs for about a year on a coin cell battery at this duty cycle.

Receivers are externally powered and consist of a pipsqueak receiver and an ALIX [2] single board computer acting as a gateway hub, pictured in Figure 5(b). The hub relays the ID, signal strength, and sensed data of each packet received to the aggregator over a wired or wireless network. The receivers have a USB interface, allowing multiple receivers to attach to each hub, as shown in Figure 5(b). Multiple receivers are required because several algorithms, such as localization and mobility detection, require signal strength samples from multiple locations.

The system supports additional radio devices, including Wi-Fi and Bluetooth, however their lifetimes were insufficient due to high power consumption. Currently these devices have only been used as proof of concept, but we are exploring ways to use Wi-Fi receivers to passively track laptops and other wireless devices.

5.1.2 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 tran-

sient solver, as described in Section 4.3.1.

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 [10]. 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 [14], 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 [12]. 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 Octopus 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.

5.2 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. Figures 6(a)-(e) show examples of these deployed sensors.

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 Octopus.

The microcontroller in the tags has an on-chip temperature sensor which we used to detect coffee brews, as shown in Figure 6(d). 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 Octopus 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, shown in Figure 6(e), 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 Octopus 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.

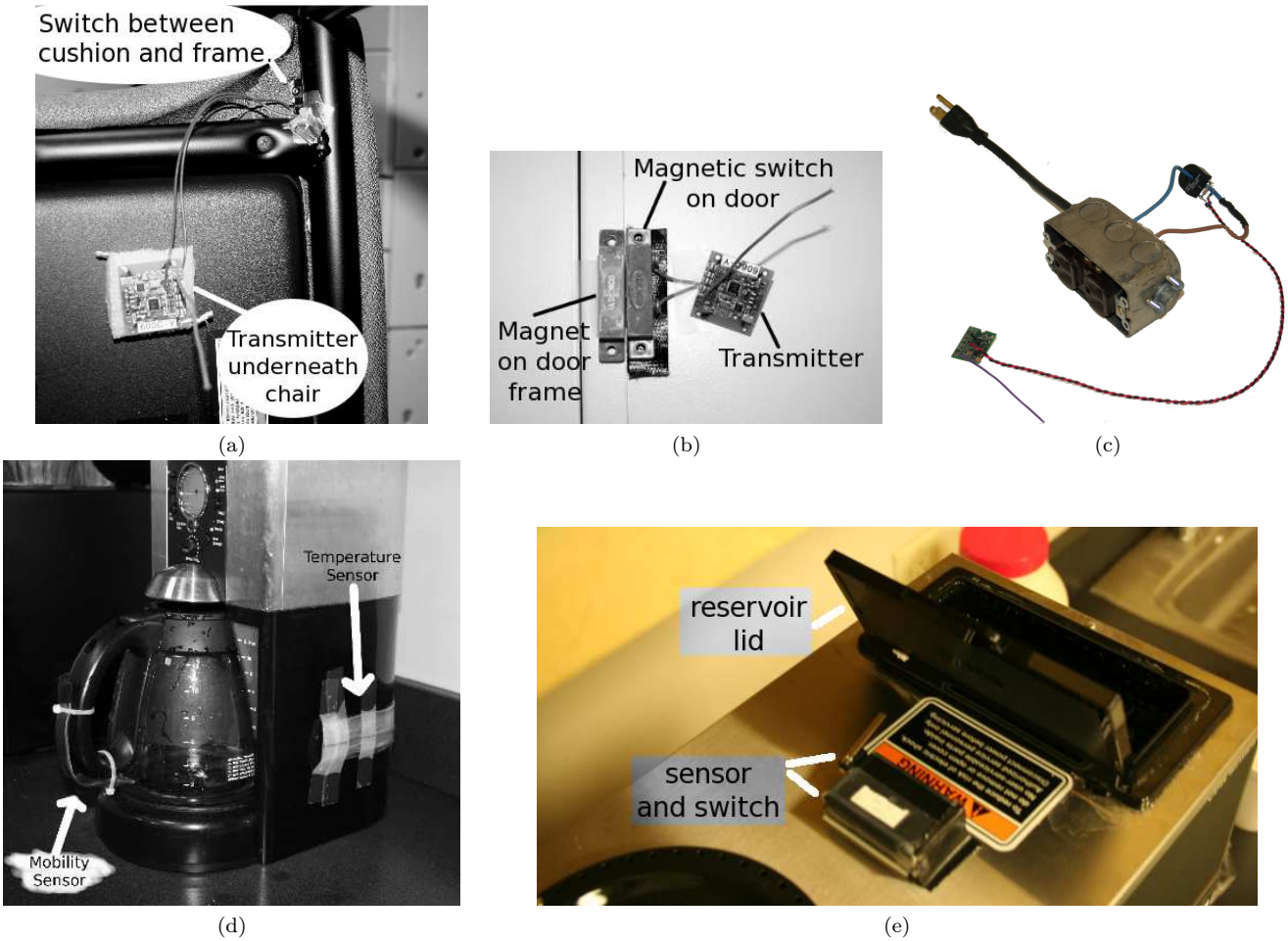
6. RESULTS: USER AND DEVELOPER EXPERIENCES

We must evaluate the effectiveness of Octopus 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.

6.1 Status Lookup and Event Updates

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, pictured in Figure 6.1(a), 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. We have tracked usage of the status map and other GUI tools since we began advertising the system to the general building populace in Figure 8.

In addition to the live status map, we also provide a "push"



6: These modified pipsqueak tags act as sensors. Figure (a) shows switch sensors used in chairs, and Figure (b) shows them attached to doors. Figure (c) shows a power sensor, which reports when power is drawn from a socket. The circuit was enclosed in the box during deployment. The coffee sensors in Figure (d) detected rises in heat when coffee was brewed and pot motion when coffee was poured. However, the coffee machine was replaced with a model that heated its reservoir so the heat sensor was replaced with a switch to sense when water was added to the machine (e).

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 Octopus 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.

Octopus users register a few events that they are willing to receive notifications for. For other events, they will check the status map manually.

6.1.1 *Finding Items*

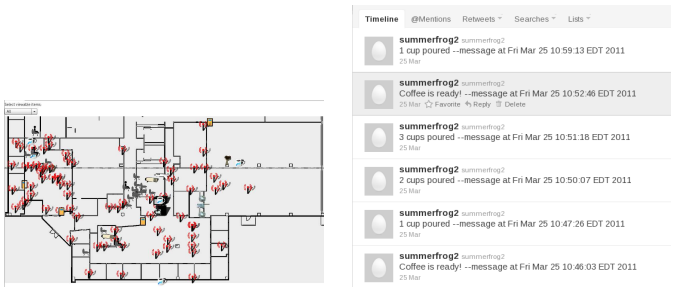
Next let us look at a specific example, finding coffee mugs, to explain how users can use the status map to retrieve information. The most common use of the map is to find missing coffee mugs. The status map, pictured in Figure 6.1(a), includes a map area that uses results from the localization solver to show the positions of various items, such as mugs, tape, a hot glue gun, and so on. Each item has its own icon so that users can quickly find what they are looking for, and a drop down box allows users to limit the items shown on the map to certain groups of items, for instance just mugs, or to a single item, for instance a particular person's mug.

To reiterate, users would begin walking with a mug and put it down at point but forget where they left them. Typically, a user would use the interactive map to locate his or her mug, then head over to the room where the mug was. Even though the localization accuracy was only good to 6-10 feet, the number of places the mug could actually be was limited enough that users had little trouble locating the mug once they knew which room it was in.

6.2 System Management

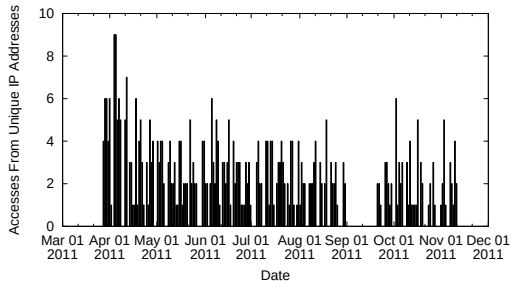
Although the system was initially deployed by experts, we invited novice users to make use of the system for personal projects, as course work, or to obtain data for their own research. We provided several management tools to help users add data into the world model.

The simplest tool simply reads information from a text file and adds it into the world model and was intended mainly



(a) (b)

7: A screen shot of a live status map (a) with all items visible. A dropdown box allowed users to control what was displayed and mouse-over tooltips allowed users to to visually inspect room use, search for items, or check the status of special items. However, users prefer "push" updates for some information, such as coffee updates, through Twitter (b) and email.



8: Unique users (as counted by IP addresses) of the GUI tools every day over several months of the deployment. This does not reflect subscriptions to email notifications. The gap from September to October was caused when the monitoring script was not restarted after a power outage shut down the computer.

to make it easy to add large amounts of information into the system at once. The second tool displayed a map of an area and allowed users to click on the map to add new immobile items. This tool was useful while deploying new sensors with fixed locations. Finally we created a general-purpose browsing and editing tool that presented a folder view of all information in the system. A screen shot of this tool appears in Figure 4

Multiple people may be changing and adding information in the world model simultaneously, providing a good test of Octopus's multi-user support. The world model's ability to track different sources of data made the system much easier to configure because we did not have to worry about someone "breaking" parts of the system with a poor configuration. When a person put in bad data we simply deleted the bad entries and the system would fall back to the previous settings.

6.2.1 Automatic Dependency Management

One important issue for novice users is confusion about the purpose of different solvers, which solvers depended upon one another, and which solvers were necessary to specific end-user applications. Generally, developers can use information in the world model without worrying about its source, but if when a person runs a stand alone Octopus system they need to at least start the solver processes. To simplify this we built solver management client that can automatically start solvers based upon settings in the world model.

For instance, a signal strength-based mobility detection solver could be named "solver.mobility detection" in the world model and could have two attributes: a vector of attributes that the solver depends upon and a vector of attributes that the solver provides. In this case a mobility detection solver might require signal strength values and provide an attribute named "mobility.boolean". Keeping track of which other solvers each solver depends upon is a form of domain-specific knowledge, so encapsulating this knowledge could prevent difficulties for novice users.

This solver management client is similar to package management systems in linux that automatically find and download package dependencies and we note that in the future the solver management client could gain the ability to download solvers from an online repository. This would give users the ability to update their Octopus systems very easily.

6.3 Low-level Developers

Our deployment has also supported several developers who interacted with the lower levels of the system. One developer used the system as a way to demonstrate their own sensor hardware. This person connected their hardware to the aggregator and reused the rest of the system. Using the existing temperature solver and the status map they were able to display temperature results from their hardware on the map.

The system has also proven useful for research. A PhD student working on a radio analysis algorithm was able to use the system as a source of data for their research. These experiences show that Octopus provides great gains in terms of software reuse and abstraction. Each of these developers was able to concentrate on a single part of the system as the Octopus aggregator and world model successfully partitioned the system into three separate views.

7. CONCLUSIONS AND FUTURE WORK

We addressed the complexity of multi-user pervasive systems and introduced a new architecture, named *Octopus*, 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.

Octopus 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 Octopus 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 3 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 Octopus systems in more locations, and in future work we hope to be able to present interesting solutions in this area.

8. REFERENCES

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

[2] Pc engines alix system boards. Website, Retrieved November 14, 2011, `http://pcengines.ch/alix.htm`

[3] 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. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Syst (2004)

[4] Aberer, K., Hauswirthý, M., Salehi, A.: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In: Mobile Data Management (MDM) (2007)

[5] Al-Jaroodi, J., Mohamed, N.: Service-oriented middleware: A survey. Journal of Network and Computer Applications (0), – (2011)

[6] Anke, J., Müller, J., SpieSS, P., Weiss, L., Chaves, F.: A service-oriented middleware for integration and management of heterogeneous smart items environments. In: Proceedings of the fourth MiNEMA workshop. pp. 7–11 (July 2006)

[7] Firner, B., Jadhav, P., Zhang, Y., Howard, R., Trappe, W., Fenson, E.: Towards Continuous Asset Tracking: Low-Power Communication and Fail-Safe Presence Assurance. In: Sensor, Mesh and Ad Hoc Communications and Networks, 2009. SECON '09. 6th Annual IEEE Communications Society Conference on. pp. 1–9 (June 2009)

[8] Harle, R.K., Hopper, A.: Deploying and Evaluating a Location-Aware System. In: Proceedings of the 3rd international conference on Mobile systems, applications, and services. pp. 219–232. MobiSys '05, ACM, New York, NY, USA (2005)

[9] Kleisouris, K., Chen, Y., Yang, J., Martin, R.: The Impact of Using Multiple Antennas on Wireless Localization. In: Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on. pp. 55–63 (June 2008)

[10] Kleisouris, K., Firner, B., Howard, R., Zhang, Y., Martin, R.P.: Detecting Intra-Room Mobility with Signal Strength Descriptors. In: Proceedings of the eleventh ACM international symposium on Mobile ad hoc networking and computing. pp. 71–80. MobiHoc '10, ACM, New York, NY, USA (2010)

[11] Koutsonikola, V., Vakali, A.: Ldap: framework, practices, and trends. Internet Computing, IEEE 8(5), 66 – 72 (sept-oct 2004)

[12] Krishnan, P., Krishnakumar, A.S., Ju, W.H., Mallows, C., Ganu, S.: A System for LEASE: Location Estimation Assisted by Stationary Emitters for Indoor RF Wireless Networks. In: Proceedings of the 22nd IEEE International Conference on Computer Communications (INFOCOM) (Oct 2004)

[13] Luo, L., Kansal, A., Nath, S., Zhao, F.: Sharing and exploring sensor streams over geocentric interfaces. In: Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems. pp. 3:1–3:10. GIS '08, ACM, New York, NY, USA (2008)

[14] Madigan, D., Einahrawy, E., Martin, R., Ju, W.H., Krishnan, P., Krishnakumar, A.: Bayesian Indoor Positioning Systems. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE. vol. 2, pp. 1217 – 1227 vol. 2 (2005)

[15] Mansley, K., Beresford, A.R., Scott, D.: The Carrot Approach: Encouraging use of location systems. In: Proceedings of the Sixth International Conference on Ubiquitous Computing (Ubicomp '04). pp. 366–383. Springer-Verlag (Sep 2004)

[16] Mark Weiser: The Computer for the Twenty-First Century. Scientific American (September 1991)

[17] Ravindranath, L., Padmanabhan, V.N., Agrawal, P.: SixthSense: RFID-based Enterprise Intelligence. In: Proceeding of the 6th international conference on Mobile systems, applications, and services. pp. 253–266. MobiSys '08, ACM, New York, NY, USA (2008)

[18] Stallings, W.: Snmpv3: A security enhancement for snmp. Communications Surveys Tutorials, IEEE 1(1), 2 –17 (quarter 1998)

[19] Willett, W., Aoki, P., Kumar, N., Subramanian, S., Woodruff, A.: Common sense community: Scaffolding mobile sensing and analysis for novice users. In: Pervasive Computing, Lecture Notes in Computer Science, vol. 6030, pp. 301–318. Springer Berlin / Heidelberg (2010)