It's Tea Time: Do You Know Where Your Mug Is?

Robert S. Moore[*]
romoore@cs.rutgers.edu

Bernhard Firner[†]
bfirner@winlab.rutgers.edu

Chenren Xu[†]
lendlice@winlab.rutgers.edu

Richard Howard[†]
reh@winlab.rutgers.edu

Richard P. Martin[*]
rmartin@cs.rutgers.edu

Yanyong Zhang[†]
yyzhang@winlab.rutgers.edu

[*]Computer Science Dept, Rutgers University, Piscataway, NJ, USA
[†]WINLAB, Rutgers University, North Brunswick, NJ, USA

ABSTRACT

The transition to Internet of Things depends on the ability to create small, simple applications that are easily written and can be flexibly combined into larger, more powerful systems. We have designed an infrastructure to meet this need and report on a year's experience expanding and using it in an open-plan academic office space with up to a hundred sensors enabling nearly a dozen applications ranging from announcing tea time in the break room, notifying users that the conference room is in use, to printing documents from a web-based map. Applications are simple to write, modular, easily reused, and can incorporate diverse data inputs in a heterogeneous sensing environment. We discuss our efforts to incrementally improve user interfaces and system management.

Keywords

Smart Building, Modularity, Reusability.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems.

General Terms

Design, Experimentation

1. INTRODUCTION

It's lunchtime so Bob walks to the other side of the building to heat his casserole in the communal kitchen. There is already a line in front of the microwave but it would be a waste of time to walk back to his office so Bob just waits outside the kitchen. Alice walks into the kitchen, looks at the busy microwave, and joins Bob.

"I wish I knew when the microwave was busy so I didn't have to spend so much time waiting in line," says Alice. Bob nods in agreement and looks at the coffee pot. "Do you know how old that coffee is?" Alice shrugs.

Bob dumps out the coffee and starts a fresh brew. "You know, we work in a technology lab, there should be a way we can monitor these things remotely." Alice pulls out her smartphone and starts its web browser. After a brief search she replies, "Some hackers have made internet coffee machines, but there's nothing on Amazon."

The microwave is finally free so Bob starts warming his food. "How about a smart microwave?" Alice shakes her head. "Maybe we could build one."

Eve walks into the kitchen, searching for something. "Hey, have you seen the duct tape?" Alice and Bob shake their heads. Eve storms out, griping "If only we had a Marauder's Map, I'd hack it to track that roll of tape instead of people."

Alice and Bob exchange a knowing glance. They know what they need to build, but how can they make such a platform for their personal application needs from scratch?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotPlanet'13, August 16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2177-8/13/08 ...$15.00.

Vision of Internet of Things

Mark Weiser proposed connecting network devices to enhance building computing systems when before in-home networking was commonplace [5]. As networking devices became more commonplace, Gershenfeld et al. proposed that increasing presence of individually networked devices would lead to an *Internet of Things* (IoT) where data from common objects in the home and office would always be available [2].

The Internet of Things Quest

To build an IoT platform, ideally we would have a system or middleware that software and hardware modules could "plug into" with little effort [1]. This would allow us to start with a small test system, for instance it could be a single sensor on the coffee pot and a small amount of code that tells Alice and Bob when coffee is brewed. The system should be flexible enough that when a power sensor is brought to monitor the microwave, it could be added into the system easily. The system must also be flexible enough to add new information, such as real-time location information for a Marauder's Map [6].

The key feature we explore in this work is the concept of organic growth applied to IoT systems. The research challenge arises because it is impossible to predict what applications people will want, and thus what sensing will be needed. In this paper we describe our experiences incrementally deploying and organically expanding our IoT system.

The rest of the paper is organized as follows. In Section 2 we outline our approach to building an IoT system. In Section 3 we detail our first steps toward building the system. In Section 4 we summarize the various components that have expanded the capabilities of the system over the past year. Finally, in Section 7 we summarize our work and describe our next steps.

2. DESIGN GOALS

We begin our discussion of the system's design goals by outlining a few basic characteristics that are desirable, practicable, and useful in an IoT system.

- **Simplicity** The system should make the fewest assumptions possible. Namely concerning hardware capabilities, data constraints, and application goals.
- **Modularity** Components should be clearly defined and separated from each other. Vertically-integrated solutions rarely achieve the amount of flexibility such a system should provide, so a layered approach will be preferred. Interfaces should be simple, only as complex as necessary to achieve their purpose, and any APIs should be designed in a language- or hardware-neutral manner.
- **Heterogeneity** The system should support a wide range of application types and goals. It should unify the different components in such a way that enables interaction between them, but never so much that it restricts those same interactions.
- **Reusability** Alongside modularity, the system should encourage developers to avoid duplicating efforts by making data and component reuse as easy as possible. In general, reusing existing data should be easier than recomputing it.

With the systems' primary characteristics or goals in mind, we will start with a thought experiment to motivate both the design and the development direction. Not only an example of what we wish to achieve with the system, the thought experiment will serve as an acid test.

2.1 The Thought Experiment

Joe is a software developer who was just assigned a new project: build an application that will allow users to quickly identify which conference rooms are vacant and which are occupied. Luckily for Joe, Alice and Bob have already deployed an IoT platform. Joe's first step, then, is to inspect the system and see what data is available about the conference rooms.

The system needs to have a centralized view of the data, so that developers can quickly inspect and determine what information is available for them. The system need not be centralized, but a unified view needs to be available. We will call this view the **World Model** and for the moment assume that it is a single entity. The World Model unifies the data from the system and presents a *simplified* view to developers who wish to produce new system components.

It is reasonable to assume that we will have a large variety of data stored in the World Model: sensor readings, images, text, building parameters, and many other kinds of data. To support arbitrary data types and relationships, we will use a very simple structure to represent information. Each point of data in the World Model will be represented by a **Identifier** (ID) and a set of zero or more associated **Attributes** that describe the ID. The ID will take the form of a "dotted-string", *e.g.*, "hallway.light bulb", "UUID.ABCDEF1234", or "My Favorite ID \.". While many types of data lend themselves to categorization or hierarchies, others do not. The decimal (.) will serve as a special value, denoting separations between the layers of an information hierarchy. At this time, both the definition and format of hierarchical data is treated as an informal convention, and is not enforced.

Joe inspects the World Model and sees that there is already a power sensor on the conference room projectors — Alice and Bob decided that they were interested in power consumption for more than just the microwave. The system is also tracking WiFi device locations so that the network operators can identify "hot spots" of activity and more effectively deploy access points. Joe decides that he can reuse *these two types of data to add an "in use" attribute to the conference rooms. He decides that if the projector is turned on, the room is* probably *in use, but someone may have forgotten to turn it off before leaving so the "in use" attribute will express a probability rather than a binary state. If a WiFi device is also present in the room, he will raise the confidence level of his new Attribute.*

The component that Joe is proposing to add, a program that will produce a new piece of information in the World Model, is called a *solver*. The system should support a plethora of solvers, each consuming and producing different types of information. Each of these solvers will effectively be a *module*, which when added to the system expands the world model. The combination of these individual solvers, producing for and consuming data from the World Model, will form a very rich and dynamic representation of the environment.

*But Joe isn't entirely satisfied with this solution: what if someone leaves the projector on AND forgets their cell phone? He needs something more definite for his project, so he decides that monitoring chair utilization will give him the data he needs. If someone is in the chair, he can be almost 100% sure that the room is in use, more sure than just with data from projector and WiFi information. With this in mind, he asks Alice and Bob to install seat pressure sensors in all of the conference rooms. They agree on a standard ID format for the new sensors, perhaps "conference room.*X*.chair.*Y*", where* X *and* Y *are unique identifiers for conference rooms and chairs, respectively. They also decide how the value of this new attribute will be represented: a single-byte value with 0 for empty and 1 for occupied. They agree that in a month the new sensors can be delivered to the first of the conference rooms and the solver written, the rest of the conference rooms will be deployed on a rolling schedule.*

Joe knows how to inspect the system, and is familiar with the idea that solvers are required to interpret the raw sensor data. Once an ID format is agreed upon, Joe can start writing his application. If the data isn't yet present, he can fall back on the existing projector and WiFi data, and once chair data is available he makes use of it. Nonetheless Joe doesn't need to know *how* the data is generated: he doesn't want to know the localization algorithm, or how the projector power is monitored, so the system should hide unnecessary complexity.

Supporting an arbitrary number and type of solvers is an important feature of the system. The simplest solvers take sensed or external data and feed them into the World Model with the necessary contextual information, *e.g.*, temperature, occupancy, location, or calendar events. A large number of these simple solvers can be produced, each independent of the other. Once these data are provided to the World Model, however, a new type of solver can be written that *consumes* these data from the World Model, performs an analysis, and pushes new data back into the World Model. It may also consume information from outside of the system. In this way, it is possible to incrementally build increasingly complex solvers that depend upon one another. Since the World Model is the interface between each of them, dependencies may be hidden and a simple model of the system is preserved.

Joe builds his prototype application and demos it to his project manager. He mentions how he is generating the confidence value, how he has written a simulator for the new chair sensors, and how the new application will automatically make use of the data once it *is available. The project is finalized and deployed for company use. After a few weeks, the new chair sensors are installed, the chair solver started, and his application starts using the data without any changes.*

Joe has taken us through what we hope will be the general use case for the system we develop. There are a few issues we need to address before we're ready to design the system. Namely, how will solvers get access to raw sensor data? How do we manage various solver dependencies in a simple and uniform way? Is a hierarchy for World Model IDs necessary? If so, how should it be enforced?

2.2 Working Out the Details

To provide a uniform way for solvers to access sensing data from a potentially wide assortment of underlying sensors, we introduce the idea of the **Aggregator**, a simple component that acts as a sink for all the sensors, and a source of data for solvers. Since few solvers will require data from all sensors present, the Aggregator should allow solvers to specify only those sensors that they wish to receive data from. A secondary advantage of this is reducing the amount of data that needs to be transmitted from the Aggregator to the Solver. The Aggregator should also remain agnostic to the types of data that it is sending. It should not provide more than a filter-and-forward service for the solvers, leaving data analysis for the higher layers of the system.

The Aggregator is important because it separates the task of adding a sensor from interpreting its data and allows for many users to easily reuse sensor data. Imagine if the system was a monolithic silo — Alice and Bob have put a power sensor on the projector because they were curious about the projector bulb lifetime. When Joe asks how he can access the sensor data, Alice and Bob regretfully tell him that reading sensor data is tied into their stand alone piece of software. Specifically, if the sensor "pushes" data, Joe will have to modify their software to receive projector data, or Alice and Bob will need to modify the sensor to send data to two applications. If data is "pulled" from the sensor then the two groups must worry about unforeseen interactions between simultaneous pull requests.

With the Aggregator these problems are easily avoided. Whether sensor data is "pulled" or "pushed" it is only read once and the Aggregator distributes it to interested parties and any number of applications can be written that use the data, without fear of interactions between them.

Over time, it is reasonable to assume that a large number of solvers will be written for the system, each making some new type of information available to the system. As the system expands and the number of solvers grows, we need to make sure that developers can easily navigate and extract the data they need from an ever-increasing world model. By utilizing the World Model as a single source of all these attributes, developers never have to worry about where their dependencies are running, how to invoke them, or the transitive dependencies involved (dependencies of dependencies).

2.3 Putting It All Together

Figure 1 provides a nicely-structured view of the different layers of the system. By keeping the layers well-defined and interfaces clear, we can hide unnecessary details and modularize the components.

At the highest level, we have the **Application Layer**, where user applications reside. The Application Layer will also include utility software that uses system data and provides it to other applications.

Next, we see the **World Model Layer**, containing the World Model. It will serve as a centralized point of contact for solvers and clients to request information about the system and the environment.

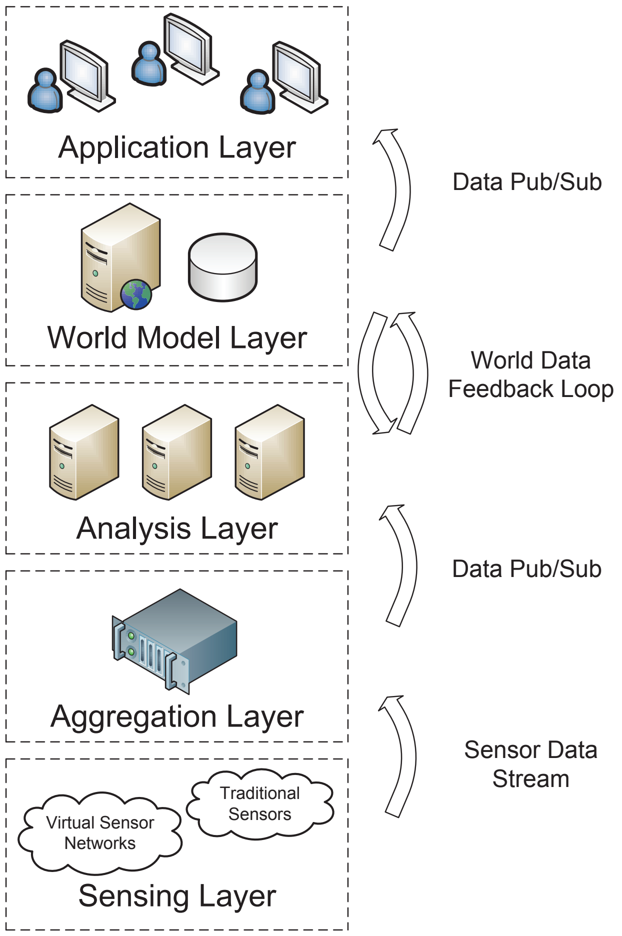


Figure 1: An outline of the proposed system architecture. All of the major components are identified and the primary data pathways are indicated as arrows.

Continuing down, a number of solvers are present in the **Analysis Layer**. The solvers here consume raw data from sensors below as well as information from the World Model, and produce new data in the World Model. By keeping each solver as simple and specialized as possible, we can maximize opportunities for reuse and efficiency.

Just below the Analysis Layer is the **Aggregation Layer**, where a unified view of sensor data is provided to solvers. The Aggregator presents a simple publication/subscription model to solvers, and performs simple filtering for the solvers to reduce the amount of unnecessary data being sent.

The lowest layer of the system is the **Sensing Layer**, where various sensors reside. The only assumption the system makes about this layer is that the sensing components are capable of providing data to the aggregator over a simple TCP/IP-based protocol.

3. INITIAL DEPLOYMENT EXPERIENCE

We began the project with the mindset that tracking objects over time was going to be integral to many of our desired applications, though we later discovered this to be a poor assumption. Location tracking based on received signal strength has a "long tail" in its error curve that makes it difficult to implement many applications that rely upon consistent accuracy. Instead, we combined location tracking with other results, such as mobility detection and simple sensing, to accomplish our goals.

3.1 Starting with Localization and Tracking

Because tracking individual items is an essential part of our system, the first version had an Aggregator, World Model, and Bayesian localization [4] solver as the only components. Our early work focused on tracking custom radio tags, whose pictures are shown in Figure 3.2(a).

Our architecture allowed us to easily add new wireless devices to track, but the initial version of the localization solver was not very flexible. In particular, we found that we wanted to use some of the intermediate information computed by the localization solver but did not want to replicate the code. For example, both passive and active mobility detection needed vectors of received signal strength; sharing these values would allow much easier implementations of those solvers.

We thus decided to break up the single localization solver into three parts: an "on-demand" solver that computes signal strength statistics, a mobility detection solver that would detect when objects had moved through signal variance [3], and a localization solver that relied on the other two. Mobility information is used as a hint to determine when localization should be performed. In this way we have the solvers build on each other, achieving the original goal of device localization, and enabling other applications to use signal strength vectors and mobility independently.

The new localization solver was triggered by actual mobility in the environment, and we immediately noticed that localization results were closer to "real time" since the system was only re-localizing objects when they moved. As a failsafe, we also programmed it to localize any object that hadn't been localized for more than five minutes, in case of errors in the mobility detection solver.

3.2 Adding Sensing

Recognizing that tracking was not going to be sufficient for all of our desired applications, we added simple infrastructure monitoring. The radio tags we chose had input pins that allowed us to attach simple sensors and also had a temperature sensor integrated with the microcontroller. Some examples of these sensors appear in Figure 3.2.

We programmed the physically modified tags and attached them to various objects around the lab: magnetic switches on doors to see when they open and close, power sensors for projectors and televisions, and programmed the on-board temperature sensor for the coffee machine. We also put a standard tag on the handle of the coffee carafe and would use mobility detection for this tag to detect when people had picked up the carafe and poured a cup of coffee.

Using the temperature and carafe tags on the coffee maker, we created a "coffee solver" which would produce the "coffee brewed" attribute with a date/time value. The brew time would be based on monitoring the temperature of the coffee pot sensor, identified as a rise above a threshold value.

Our next change occurred when the original coffee machine broke down and was replaced by one that no longer enabled easy temperature monitoring. Because of the popularity of brew notifications, we devised a new technique where a switch was attached to the coffee maker sensor that would be triggered whenever the user loaded it with fresh water. The coffee solver was changed to update the brew status whenever the switch was triggered, with a 20-minute cool-down period to avoid double counting. This transition was seamless to the applications that used data from the coffee solver

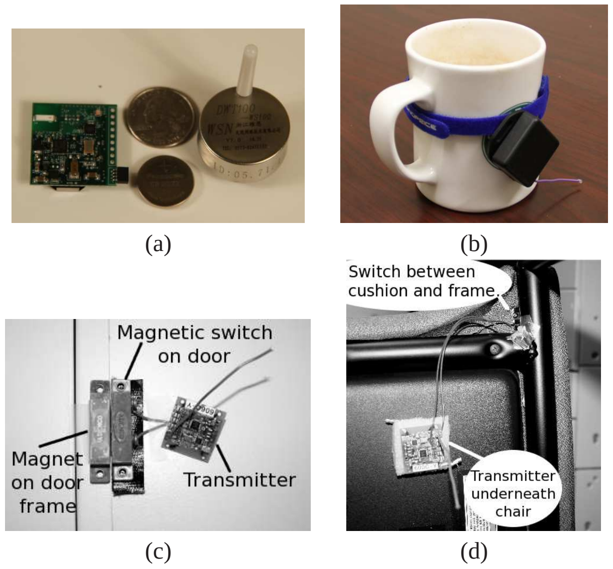


Figure 2: (a) shows the relative sizes of the sensor we are using for most tracking and monitoring (on left) and a different sensor we used for its water and heat resistant packaging (on right). (b), (c), and (d) show our sensor deployed to track a mug, detect when a door opens or closes, and detect when a chair is in use.

– updates before and after the change were indistinguishable to the higher layers of the system. The switch-based approach allowed us to use the same solver to detect coffee brew events, door open/close events, chair use status, and other information that had a switch sensor. This solver was about 140 lines of C++ code.

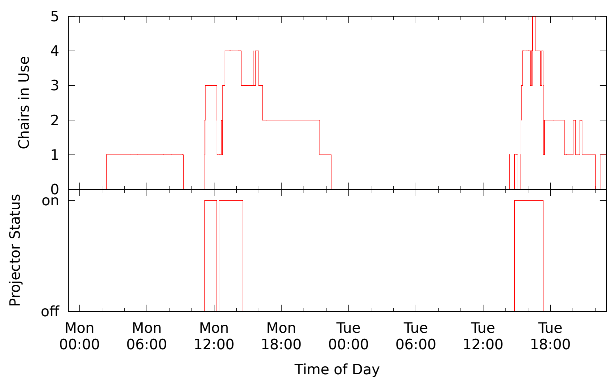
Encouraged by the integration of the coffee solver with our daily activities, we developed another solver to help automate a lab social event. Some afternoons, members of the lab meet in the shared kitchen to socialize over some coffee, tea, and snacks. We called this "Tea Time" and it continues to be a popular event. However, it is an ah-hoc, unscheduled affair, that depends on who is in the lab and if snacks are available. Before tea-time occurs the people who want to initiate the gathering must go through the troublesome process of gathering everyone to the kitchen. To make this process easier we wanted to write a solver to detect when tea time occurred so that we could notify people automatically.

We didn't want to write single-purpose solver just to detect tea time, so we wrote a generic solver, called the *gathering solver*, to detect gatherings of different kinds of items in defined regions. Tea time was then defined as three or more mugs localizing to within the kitchen area between 2 and 5pm on a weekday. This solver was written in about 130 lines of C++ code.

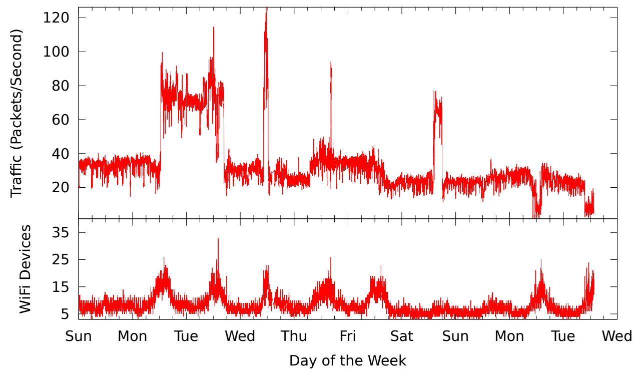
4. APPLICATION DEVELOPMENT

In the previous section, we detailed how the system grew in the early stage of deployment. In this section, we will describe a small sample of additional solvers and applications that were added to the system over time. These solvers are diverse in nature and intended to serve different user groups. This experience has demonstrated that the design of our system has made an application developer's job rather easy.

WiFi Device Monitoring: In preparation for a second system deployment which centered around WiFi (IEEE 802.11) devices, we



(a)



(b)

Figure 3: Data and screenshots from conference room usage (a) and WiFi monitoring (b).

added WiFi radio cards to some of our receivers and a solver that monitors the number of active devices throughout the day. Every minute it samples a WiFi channel for 20 seconds and records the number of unique MAC addresses and traffic level measured in packets per second. Due to user privacy concerns, we are not recording device addresses or tracking specific devices. Counting the number of active WiFi devices, however, has allowed us to identify periods of high WiFi utilization or WiFi failure, as shown in Figure 3(b). As the next step, we plan to combine this with anonymized location information for smartphones to identify spatio-temporal regions of high utilization, which we believe will enable a large array of applications that center around presence detection and event detection.

Passive Localization: Tracking human subjects is of interest to many potential applications. Our system offers several options for this purpose. As mentioned above, it can track a person's smartphone through the WiFi monitoring solver, or it can tag and track a person's coffee mug through the active localization solver. In addition, we also implemented a device-free passive localization and counting solver [8, 7] that find how many people and where are they based upon their impact on ambient radio signals. Passive localization can trace people without revealing the identities of them, and is suitable for applications with strong privacy concerns. In the 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 [9] or off-the-shelf smartphones to bring more sensor modalities to extend the solver to track more people.

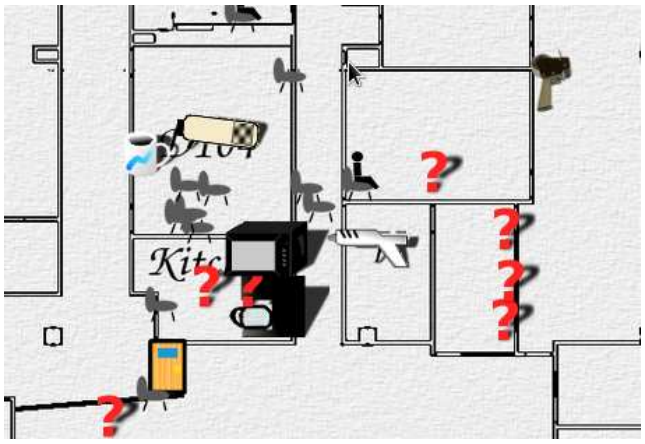


Figure 4: A portion of the HTML map that we developed to replace our early Flash-based map. We have removed option panels and other features and focused on a subsection of the map to make details clear. Question marks indicate objects which are unknown to the map.

Conference Room in Use: This is the implementation of our original thought experiment. We added chair sensors to all the chairs in two different conference rooms in our laboratory/office space. We combined this information with a power sensor connected to the projector or television in the room, and reported that the room was occupied when the projector was on and at least one chair was occupied. An example of this data for one of our conference rooms appears in Figure 3(a).

The data provided by this solver shows details of the conference room that may be useful to the people managing the lab space. For instance, the single chair used between 2 and 8 AM might imply that someone slept in the lab that night so a couch in a private area might be a good addition to the lab. We can also see that there were two distinct meetings between 11 AM and 2PM because there was a drop in chair use and the projector turned off at noon. However, one chair remained in use over the course of both meetings so there was probably a person who attended both.

Marauder's Map: We first built a simple Flash-based live status map that showed a map of the lab space and the locations of each localized object as an icon displayed on the map so that users could see where objects were in real time. The map shows the real-time locations and status on the map. Status was indicated with different icons: doors opened and closed in real time, projectors lit-up, TV screens flickered to life, etc. We found that requiring flash was actually limiting user adoption so we reimplemented the status map as a Ruby generated HTML page in just over 400 lines of code, pictured in Figure 4.

The map provides an easy way to check for conference room usage since chair icons change to indicate if a person is sitting in them and the projector shows a light to show that it is currently on. Microwave status is displayed in a similar fashion, so Alice and Bob would be pleased with the system.

5. USER INTERFACES

For a laboratory-wide system like ours, user interfaces are of critical importance. On the road of getting our system out to more users, we have made a considerable amount of effort to improve our user interfaces.

The map is a very natural user interface for status information that can be seen “at a glance,” such as the usage of conference rooms or the location of a lost mug. The map interface also revealed some weaknesses in our assumptions – we actually noticed that the chairs “migrated” around the office, making their occupancy status much less useful for determining the conference room status as they ended up in entirely different rooms. This lead to several “chair hunts” to find the instrumented chairs and bring them back to the conference rooms.

Although the map itself is useful for some results, it is not suitable for some other applications such as “fresh coffee” – users had to “hover” their mouse over the coffee pot icon on the map to see a tooltip value with the last brew time and cups poured. To make the system more attractive to coffee lovers we decided to implement a push-based notification system. We set up a mailing list that users could subscribe to, and we also created a Twitter account, both for the coffee pot in our lab. To make signing up for updates as easy as possible we deployed QR codes as shortcuts to subscribe to the coffee mailing list.

6. SYSTEM MANAGEMENT

We have developed two utility solvers that can help us manage the system. The first one is a packet loss monitoring solver that can track which packets were heard by which receiver(s). This solver can help us quickly identify malfunctioning equipment and avoid radio “dead zones” for better localization accuracies.

Another utility solver is the solver monitoring solver. As the system grew more complex, we recognized the need for automated system monitoring and failure reporting. To that end, we wrote a simple solver that would look at running processes on our solver hosts and report when solvers had crashed or otherwise stopped working. We recognize that this monitoring capability is currently quite limited, and are working to expand it into a more comprehensive system management utility with a web-based interface.

Our deployment has expanded significantly over the two years of its deployment. Throughout all of these changes, the only systemic failures were when the entire building housing our servers lost power (including emergency power). For all the other situations, the sensors, receivers, Aggregator, and World Model server continued to operate normally. When components failed, their dependent components stopped seeing events, but always degraded gracefully. When repairs or replacements were made, downstream components started working once again.

7. CONCLUSIONS

Over the course of our two year deployment, we built an IoT system that was able to grow organically with users’ needs. Our decision to provide a single World Model was very effective. The simple interface it provided, as well as the flexible data model used, enabled the re-use of information in the system. Combined with a modular approach to solver development, we were able to create complex applications by layering and combining simple components. Our choice of narrow, network-based APIs, combined with API interface libraries in multiple languages gave developers the freedom to choose what worked best for their component development. As the number of sensors grew from 1 to nearly 100, the system was able to continuously operate without error, and scaling was not an issue.

8. REFERENCES

[1] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *VLDB*, 2006.

[2] C. D. Gershenfeld N, Krikorian R. The internet of things. *Scientific American*, 2004.

[3] K. Kleisouris, B. Firner, R. Howard, Y. Zhang, and R. P. Martin. Detecting intra-room mobility with signal strength descriptors. In *ACM MobiHoc*, 2010.

[4] D. Madigan, E. Einahrawy, R. Martin, W.-H. Ju, P. Krishnan, and A. S. Krishnakumar. Bayesian indoor positioning systems. In *IEEE INFOCOM*, 2005.

[5] Mark Weiser. The Computer for the Twenty-First Century. *Scientific American*, 1991.

[6] J. K. Rowling. *Harry Potter and the Prisoner of Azkaban*. 1999.

[7] 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.

[8] 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.

[9] 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.