SmartSense

A System For Ambient Sensing at MIT

Sami Alsheikh, Feras Saad, Nick Uhlenhuth
{alsheikh,fsaad,nicku}@mit.edu
Supervisor: Prof Arvind, Section 5

May 8, 2015

1. Outline

Collecting and monitoring data about environmental conditions, such as temperature, humidity, radiation, and water pressure, is of key interest to the Facilities Department at the Massachusetts Institute of Technology. This task is now achievable due to the recent boom of inexpensive, low-power sensors that are capable of basic computation and communication. This paper proposes a design for a campus-wide ambient sensing system at MIT that is simple, reliable, and scalable.

The basic design is composed of three modules: *sensors* will be deployed at various locations around campus, and relay their readings to a *facilities central server* (FCS) via *smartphone* intermediaries. This is achieved by designing an API for sensor-smartphone and smartphone-server communication protocols. A SQL-like data model exists on the FCS, which permits highly flexible querying of the sensor data.

The first half of this paper will will describe the sensor, smartphone, and central server modules, as well as the communication protocol API between each module. The second half is devoted to analyzing and evaluating our system, and outlining relevant metrics and trade-offs that underlay our key design decisions.

2. System From High Level

After facilities install sensors across campus, each sensor collects readings of a particular type. Transferring sensor data to the centralized server is achieved via crowdsourcing; members of the MIT community carry smartphones which are registered to interact with sensors via BLE[1]. Figure 1 shows the flow of information between system modules.

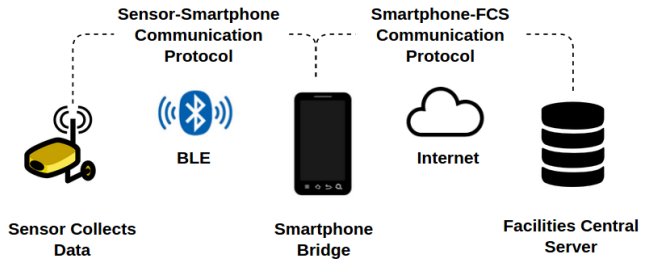

Figure 1: Overview of System Modules and Communication. The sensor collections readings from the environment, which it relays to a central server through a smartphone intermediary.

The goal of the system is to allow managers of the FCS to monitor sensor readings, especially anomalies such as smoke or high carbon monoxide levels. The system should also support queries; a smartphone can ask the sensor for latest readings, or query the FCS for archived data.

3. Sensor

The most important questions regarding sensors are determining how they should be named, how they should manage sensor readings in memory, and how to make decisions about transmitting data to smartphones. A main design consideration is maximizing sensor battery life.

3.1. Naming

Each sensor is uniquely identified by a 148-bit sensor id (sid) which is included its advertisement messages. The purpose of the sid is to encode the exact location of the sensor on campus, and the type of sensor (light, temperature, etc). The Department of Facilities maintains highly detailed floor plans of every MIT building[2]. For example, E14-790C means room 790C (on the seventh floor) of building E14. It is assumed every sensor is placed in an addressed location.

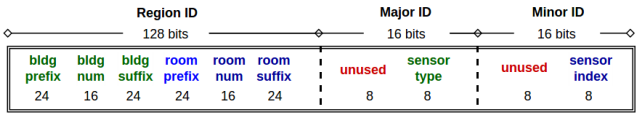

Figure 2: Naming Scheme. The Region ID encodes the sensor's location on campus, the Major ID indicates the sensor type, and the Minor ID encodes the index of a sensor at a particular location.

Figure 2 shows how every sensor in the system can be uniquely named and identified. The Region ID encodes the exact location, where prefixes and suffixes are 3 bytes that represent 3 ASCII characters. The Major ID holds the sensor type as an 8 bit number, and the sensor index is used to differentiate sensors of the same type and location.

3.2. File Management

Because sensors record large amounts of data, it is important to create a simple yet powerful data storage model. Sensors store readings in files that are created every five minutes. More specifically, at every five minute timestamp, one file is created for *normal* data and another is created for *anomaly* data. These two files are named using the format (*timestamp, isAnomaly, clockcount*), where *timestamp* is the real world time when the file was created, and *isAnomaly* indicates the type of readings stored in the file (0 for normal, 1 for anomalous). The importance of the *clockcount* field is detailed in Section 3.3.

[1] Bluetooth Low Energy is a wireless personal area network technology

[2] Accessible with MIT certificates at `https://floorplans.mit.edu`

Each sensor samples data at 1Hz. If the reading is an anomaly, it is recorded into the appropriate anomaly file. However, if the data is normal, the sensor only records the average normal reading in the subsequent 10 second interval. This averaging induces an effective 0.1 Hz sampling rate for normal data. Furthermore, averaging normal readings greatly reduces the size of the data files. If a file is empty after a 5 minute period, it is deleted to save space without deleting data. For example, suppose the normal and anomaly files are created at 15:00:00. If there are no anomalies between 15:00:00 and 15:05:00, then the anomaly file is empty and will be deleted.

The vast majority of files will only be deleted when the sensor receives an `ACK` from the FCS through a smartphone. This `ACK` specifies the name of the file that can be cleared from the sensors storage (see Section 6.3).

Each sensor reading is 6 bytes (4 byte timestamp and 2 byte reading value), which means a sensor can record readings at 1 Hz for approximately 16 days before the 8 MB capacity is exhausted. In the rare case that storage is full before connecting to a smartphone, the oldest *normal* file will be deleted; *anomaly* files are not to be deleted to make space for new data.

3.3. Clock Drift

Because the sensors might experience clock drift, we allow mobile devices to correct the sensor's time. This correction is triggered through a simple protocol outlined in Section 6. Since readings from a given sensor are keyed by timestamp, there are problems associated with correcting a sensor's time. For example, if the internal clock is reset to an earlier time, there is the possibility of having multiple data readings with the same timestamp. To better contextualize sensor data, we introduce a *clockcount* field for each sensor. Every time a sensor updates its time, it also increments *clockcount*. This scheme allows us to understand when readings with different *clockcount* values were taken with respect to each other. In the case where readings are taken with the same *clockcount*, we can differentiate between them using the *timestamp* field.

3.4. Transmission Behavior and Custom BLE Beacons

A sensor can only transmit data files when connected to a smartphone via BLE. A key challenge in our design is ensuring a sensor is not establishing connections with smartphones when it is not backlogged, since this behavior would waste power.

One hallmark of our design is employing a novel scheme where we encode the backlog status of the sensor *in its advertisement beacon*. In particular, the BLE specification allows custom data to be added to the advertising packets (alongside other connection parameters) [1]. Without delving into the technical details, we populate the `service_data` bits of the beacon packet with a 1 if the sensor has files to send, and 0 otherwise. When a smartphone receives a sensor advertisement with a 0 in the `service_data` field, it shall not establish a connection. Otherwise, the smartphone knows the sensor is backlogged and connects to receive the data file.

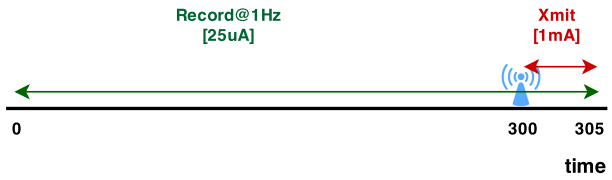

Figure 3: Power Consumption in Typical Five-Minute Transmission Window. Only a small fraction of the window is dedicated to transmission, which is much more expensive than recording and broadcasting.

The transmission cycle is outlined in Figure 3. Note that the sensor is always broadcasting its existence at 1 Hz. The blue icon indicates that the beacon now contains the flag which notifies smartphones that the sensor has files to send. When a smartphone first connects to a sensor, all non-active files are transmitted to the mobile device. More specifically, the sensor transmits all files (*normal* and *anomalous*) that have timestamps strictly less than the timestamp of the current file. While the mobile device is still connected to the sensor, only non-active files are sent (at a rate of two files per five minutes). We considered sending non-anomalous data at a reduced rate. However, this increase in complexity was not worth the negligible gain in battery life (Section 7.1).

4. Smartphone Bridge

The smartphone application will allow the user to view current connections and send queries to the FCS. When operating in background mode, the application serves as the link between sensors and the FCS. When operating in foreground mode, the application allows the user to interact with nearby sensors, or query archived data from the FCS.

4.1. Background Application

The background application is running as long as the mobile device is on. Once a sensor connection has been established, a smartphone will receive sensor readings from each non-active file on a connected sensor. The application will allow a mobile device to connect to up to ten sensors, though we do not anticipate this cap will be reached regularly. The smartphone transmits sensor data to the FCS once it has a WiFi connection. It then waits for an acknowledgement from the FCS that the data has been

received, and notifies the sensor to delete the transmitted files through the `ACK` protocol (details in Section 6.3).

The `ACK` message's `current_time` parameter is recorded from the mobile devices current time at the moment the message is sent. We assume the phone's current time is synchronized through the mobile service provider. If the phone's time is configurable by timezone, we ensure that the phone is set to EST.

The file transmission and acknowledgement will not always work smoothly, so we introduce a fault-tolerance scheme to improve robustness. In the event a smartphone dies or loses connection at any point after receiving sensor data, the application will relay the data to the FCS as soon as the smartphone is able to. If the smartphone is now unable to notify the sensor to delete the transmitted files (because the sensor is out of range), the phone notifies the FCS which sensor and files have missed their acknowledged.

When a phone connects with the FCS for the first time in the day, the FCS sends a list of sensors and messages to relay to those sensors. The mobile application will parse this message and storee all sid and message pairs, abstractly in the form $(\langle sid_1, m_1\rangle, \ldots, \langle sid_k, m_k\rangle)$. Once a connection is made with a sensor, the application will check its table to see if there are any pending messages to send to that sensor. If one of these messages is delivered, the application notifies the FCS.

4.2. Foreground Application

When the user opens the application, he will see any current BLE connections to sensors, and icons allowing him to query nearby sensors, report a problem, ping the FCS database, and refresh the view. These buttons are shown in Figure 4a.

The application derives current connection sensor information from the sid. The location is directly determined from the floor plan mapping in the Region ID, the index can be read from the Minor ID, and the sensor type can be extracted from the Major ID.

Since all sensors are constantly advertising, all nearby sensors can be stored in a drop-down list on the Landing Page. To obtain the latest reading of a sensor, the user simply needs to select the desired sensor from the drop-down list and initiate the `READ_LATEST` protocol presented in Section 6.3.

The user can choose to ping the FCS for sensor data if he logs in and waives his anonymity. This is so that the FCS can evaluate the user's permissions. Note that the user only reveals his identify if he would like to send a request to the FCS. The Ping FCS UI shown in Figure 4b allows the user to submit a request for information from the FCS. A SQL generator will be used to produce the appropriate syntax to send to the FCS for execution.

(a) Landing Page (b) Ping FCS Page

Figure 4: Wireframe Schematic for Smartphone Application. The landing page allows users to interact with nearby sensors. The Ping FCS page allows users to obtain archived data from the central server.

5. Facilities Central Server

The FCS has three roles: store sensor readings, issue configuration commands to the sensors, and reply to queries from smartphones for data statistics.

5.1. Data Model

A single reading from any sensor in the system arrives to the FCS as a 4-tuple (s, v, a, t) where s is the 148-bit sid, v is the 16-bit reading, a is a boolean variable for anamolous reading, and t is the UNIX timestamp at which the reading was originally recorded (a and t are easily extracted from the `filename`). A simple SQL database will store incoming readings with the schema as shown in Table 1.

Duplicate data is avoided by enforcing the `UNIQUE` constraint on the columns (`timestamp`, `sid`, and `clock_count`). If a smartphone sends a reading where these three fields match an existing record in the table, then a duplicate has been found, and the reading is ignored. The `clock_count` field is used to account for clock drift (Section 3.3), since rewinding the sensor clock may cause a conflict with old readings with the same (`timestamp`, `sid`). Readings with the highest `clock_count` have the most accurate timestamp.

Almost any kind of query can be performed using SQL commands; below are some examples that demonstrate the power of the database schema.

Find the average humidity on the second floor of building 10 during December 2014

Column	Data Type
timestamp	UNIX_TIMESTAMP
sid	BINARY(8)
clock_count	INTEGER(2)
sensor_type	BINARY(8)
sensor_reading	BINARY(16)
anomaly	BOOL
bldg_pre	CHAR(3)
bldg_no	INTEGER(2)
bldg_suf	CHAR(3)
rm_pre	CHAR(3)
rm_no	INTEGER(2)
rm_suf	CHAR(3)
sensor_index	INTEGER(2)

Table 1: FCS SQL Database Schema. The first three columns are primary keys, while the rest of the columns are metadata used for powerful querying.

```
SELECT AVG(sensor_reading) FROM fcs
WHERE
   (timestamp BETWEEN <2014:12:01:00:00:00>
       AND <2014:12:01:00:00:00>)
   AND (sensor_type  = <humidity>)
   AND (bldg_no = <10>)
   AND (rm_no BETWEEN <200> AND <300>)
```

Find maximum temperature on West Campus, between midnight and 6 am during the first week of August 2014

```
SELECT MAX(sensor_reading) FROM fcs
WHERE
   (timestamp BETWEEN <2014:08:01:00:00:00>
       AND <2014:08:07:00:00:00>)
   AND (DATEPART(hour,timestamp) >= <0>
       AND DATEPART(hour,timestamp) <= <6>)
   AND (sensor_type  = <temperature>)
   AND (bldg_pre LIKE <%W%>)
```

5.2. Communicating with Sensors via Smartphone Bridge

There are two situations in which the FCS must communicate with a sensor. First, the FCS can *reconfigure* a sensor, such as changing its threshold value or reading frequency. Second, the FCS knows about sensors which never received an `ACK` for their transmitted files, as discussed in Section 4.1. To remedy this situation, the FCS sends `MISSED_ACKS` messages to smartphones upon initiation of the TCP session at the start of the day. The protocol for these messages is clarified in Section 6.2.

5.3. Identifying Malfunctioning Sensors

There are two ways that the FCS can identify malfunctioning sensors. If a particular sid reports a very high number of anomalies under regular conditions, this indicates the sensor may be faulty. The second way is by cross-validating readings; if the *name* of two sensors s_1 and s_2 differ only by the `sensor_index`, then we know these sensors are in the same location and of the same type. If readings differ significantly then something might be wrong with one of the sensors, or a sensor may have been moved. A simple front-end to the database can facilitate both these tasks.

6. Communication Protocols

This section outlines the communication protocol between the FCS and smartphones (over internet), and between smartphones and sensors (over BLE).

6.1. Smartphone to FCS

When the mobile device has a file to send to the FCS, it will transmit the sid of the sensor sending the data, the filename, and the data itself. Each data entry has a timestamp t and value d.

```
DATA <sid> <filename> <t_0 d_0 ... t_n d_n>
```

As described in Section 4.1, a smartphone may lose BLE connection with a sensor before it notifies the sensor that the FCS has received its file. In this case, the smartphone notifies the FCS about this situation.

```
MISSED <sid> <filename>
```

The FCS uses the smartphone as a bridge to send `MISSED_ACKS`, `THRESHOLD`, and `FREQUENCY` messages to a sensor. If a smartphone manages to relay one of these messages to its associated sensor, it relays this success to the FCS.

```
SERVICED <sid> <original_command>
```

The only user-triggered message is a data lookup. The mobile application processes a user query for information and produces the corresponding SQL query. It sends a message to the FCS with the user Kerberos and SQL query.

```
LOOKUP <kerberos> <sql_query>
```

6.2. FCS to Smartphone

Once the FCS has received a `DATA` message from the mobile device, it sends the mobile device an acknowledgement of receipt.

```
ACK <sid> <filename>
```

The FCS will send out a message to all mobile users once a TCP connection is initiated for a given day. These messages include `MISSED_ACKS`, `THRESHOLD`, and `FREQUENCY`.

The mobile application will parse the aggregated `MISSED_ACKS` message and create entries

$(\langle sid_1, m_1\rangle, \ldots, \langle sid_k, m_k\rangle)$ in a hash table, which maps a sensor to messages meant for that sensor.

`MISSED_ACKS <sid_0 f_0 ... sid_n f_n>`

The `THRESHOLD` and `FREQUENCY` messages are both desired configurations created by an administrator on the FCS, and are meant to be relayed to the sensor. These are batched messages, which reduces the network overhead of sending individual messages for each sensor.

The `THRESHOLD` message gives the lower and upper bounds on non-anomalous data.

`THRESHOLD <sid_0 lower_0 upper_0 ... sid_n lower_n upper_n>`

The `FREQUENCY` message gives the desired time interval in seconds between *normal* sensor readings.

`FREQUENCY <sid_0 interval_0 ... sid_n interval_n>`

6.3. Smartphone to Sensor

Nearly all messages sent from the mobile device to the sensor are those forwarded from the FCS. The major difference is that sensor IDs are removed and information is sent to a given sensor one at a time.

`ACK <filename> <current_time>`

`THRESHOLD <lower upper>`

`FREQUENCY <interval>`

Finally, the only message sent coming from a mobile device to the sensor that is not simply forwarded from the FCS is the `READ_LATEST` message.

`READ_LATEST`

This message is sent by a user in order to discover the latest reading that the sensor has recorded. In the foreground application (Section 4.2), the user can select any sensor within BLE range and send the `READ_LATEST` message. The sensor will simply compare the timestamps of the most recently recorded anomalous and normal readings, and return the reading with the latest timestamp.

6.4. Sensor to Smartphone

Sensor readings are the only messages sensors will send to the smartphone.

`DATA <filename t0 d0 ... tn dn>`

7. Analysis and Evaluation

In this section, we will evaluate various metrics relevant to our system, and use calculations to justify our design decisions. In particular, we will look at the expected sensor and mobile battery life, storage requirements on sensors and the FCS, fraction of data that will be successfully transmitted over one week, and the expected time for an anomaly to be reported. We will study system components that limit scale, and features of our design that enable reliable fault tolerance.

7.1. Maximizing Sensor Battery Life With Custom Beacons

The transmission scheme outlined in Section 3.4 is designed to maximize the average battery life of a sensor while regularly transmitting sensor data to the FCS. We estimate that a sensor in our system will last roughly four and a half years.

The estimate is performed by calculating the power burned in a five minute (300 second) transmission window. When recording and broadcasting at 1 Hz, the sensor draws 25 μA, which is 0.002 mAh over 300 seconds. At the end of the transmission window, the sensor needs 1 second to connect to a smartphone, and at most 4 seconds to transmit the two files (maximum total size 2 KB, see Section 7.3) at a rate of 4 KB/sec. This transmission draws 1 mA over 5 seconds, or 0.001 mAh. It follows the total power burned in a typical five minute window is 0.003 mAh. Since the battery capacity is 1600 mAh, the sensor lifetime is 460800 five-minute windows, or 4.5 years.

Suppose a sensor in a large hall such as 10-250 is advertising using the custom beacons from Section 3.4. Several smartphones will likely overload the sensor and establish connections when the data file is ready at the and of the transmission window (at the blue icon in Figure 3). However, sensors in such crowded areas are not expected to be backlogged for long since `ACK` messages from the FCS will arrive rapidly and the `service_data` bit returns to 0. The overload effect is transient.

On the flip side, a sensor in a rarely accessed location will typically be backlogged, and broadcasting with `service_data` mostly 1 is desirable since the probability of overloading is small. This behavior does not have any negative on battery life, since broadcasting in BLE is a very cheap operation.

A key trade-off when considering battery life is the frequency of transmission. Transmitting less frequently means the sensor must establish less connections in total, which saves power. However, the size of the data files increases, which means the transmission time per connection is longer, and the FCS must wait longer before learning of anomalous readings.

We decided to transmit *both* anomaly and non-anomaly files at the five minute mark (even though non-anomaly readings are recorded at one tenth the frequency), for two reasons. First, this behavior is much simpler than having multiple transmission rules based on file type. Second,

the marginal cost of transmitting an additional 0.18 KB is small because the overhead of establishing a connection has already been realized.

7.2. Smartphone Battery Life

The MIT community will not participate in SmartSense if the application proposed in Section 4 drains smartphone battery too quickly. According to our calculations, running our application will require about 3% of the smartphone's battery life.

To assess our application's battery usage, we make a few assumptions. Because the average user may use the foreground application no more than a few times a day (less than 10 minutes total), we can assume that battery consumption in foreground mode is negligible. We focus on the background application's WiFi and BLE usage. Based on Android documentation [2], we assume WiFi draws 31 mA when transmitting, Bluetooth draws 1 mA when on but not transmitting, and that an active Bluetooth connection requires 13 mA. We assume the WiFi upload speed measured from a somewhat congested network is 1 Mb/sec. We do not require any change in a user's WiFi usage as long as they eventually connect. On the other hand, our system does require a user to always enable Bluetooth.

Consider the average user that passes and connects with 10 backlogged sensors each hour when traveling through campus. If on campus for 10 hours a day, the user will receive about 0.3 kB worth of data around 100 times, assuming anomalous data occurs about 5 percent of the time. The student is transmitting about 30 kB through BLE and WiFi. Including acknowledgement packets, the total charge required for network and BLE transmissions is no more than 0.2 mAh. The majority of our power consumption comes from the necessity of Bluetooth. This scheme pulls 1 mA and requires an additional 24 mAh throughout a whole day.

Most phones have batteries rated at greater than 700 mAh. Our application requires about 3% of most phone battery capacity.

7.3. Storage Requirements on Sensor and FCS

A sensor reading is 48 bits long (32 bit timestamp, 16 bit reading), which means an anomaly file is $300 \times 48/8 = 1.8$ KB, and a normal file is 0.18 KB. An important characteristic of *flash* storage is *block erasure*: an entire block of data must be completely erased before writing new data (we cannot append to blocks). Typical block sizes range from 512 B to 2 MB (i.e. $2^{\{9,10,\ldots,21\}}$ bytes). We shall assume the sensor's 8 MB flash device has a block size of 64 KB, so only 128 writes can be performed before we need to rewrite blocks. Our solution to this limitation is to hold batches of data files in the 64 KB RAM, and perform bulk writes (around 32) to an entire block in flash storage.

We now consider the FCS storage requirements. The typical sensor records 580 KB of readings per day. If we have 10,000 sensors around MIT campus, the data flow is 580 MB per day, or 185 GB per year. If a sensor reading is one tenth the size of a typical database entry (see Table 1 in Section 5.1) the FCS database grows by roughly 2 TB every year.

7.4. Proportion of Delivered Data

Due to our robust scheme, we expect that the vast majority of collected data will ultimately be delivered to the FCS. Because a sensor continues to retransmit its readings until it has received an ACK, the only case in which data would not ultimately make it to the FCS involves abandoned sensors. If a sensor is never visited by a mobile device, those readings will never be received. If a sensor continues to store 2 KB every five minutes on its 8 MB memory, it will be able to last for over 16 days before overwriting old data. This circumstance is highly unlikely, so we conclude that a negligible amount of data will not eventually arrive to the FCS.

7.5. Expected Transmission Time

The transmission behavior detailed in Section 3.4 is designed to quickly deliver both anomalous and normal data to the FCS. To illustrate this, recall that *anomaly* and *normal* files are ready to be transmitted every 5 minutes. In the best case, a phone is connected, immediately receives the files from the sensor, then transmits them to the FCS. The average time it takes for a reading to be reported to the FCS is then $5/2 = 2.5$ minutes. However, there will not always be a user in a location that is experiencing an anomaly. It is expected that the average sensor will be connected with about once per hour. This assumption is made on the basis that classes end every hour, and that students and faculty will be roaming around campus more frequently at these times. For sensors that only receive smartphone connections once per hour, the average time for anomalous and normal readings to be reported to the FCS is 30 minutes. While 30 minutes may seem like a long time, it is important to note that the transmission frequency is limited by the popularity of the sensor location. There is no way to transmit data without having a smartphone in range.

7.6. Potential Factors That Limit Scale

Our design is highly scalable and there are no major threats to scalability. The sensor naming scheme (Section 3.1) allows for 2^{128} locations, 2^{16} sensor types, and 2^{16} sensors of the same type in the same location. These bounds are much larger than necessary to accommodate a campus-wide system, and allows the system to grow without naming collisions.

Another potential limiting factor is sensor storage; however, as outlined in Section 7.3, sensors can record files for 16 days before running out of storage which is more than enough to satisfy the system requirements. Similarly, Section 7.3 reveals that the FCS only grows by roughly 2 TB every year. This growth is reasonable, especially considering that not all data from previous years will need to be kept in storage, and even premium 2 TB hard drives are roughly $200 in today's market.

7.7. Fault Tolerance

The system is very tolerant to WiFi connectivity failures. This resilience comes from the smartphone behavior when receiving data from a sensor through BLE. After data has been received, it is stored locally until a WiFi connection has been initialized by the user. At this point, the data will be transmitted. This scheme implies that as long as the smartphone user eventually connects to a network and the device does not run out of storage, our system will work as intended.

Another way the system achieves fault tolerance is through *replication*. As discussed in Section 3.4 a sensor only deletes data from storage upon receiving an ACK from a mobile device. The process works as follows: a sensor will send a file, *file A*, to all phone connections until it receives an `ACK` saying *file A* can be removed from storage. Because this replicated data is sent to several phones, there is a higher likelihood that at least one of the phones will be able to connect to the FCS and send the appropriate `ACK` back to the sensor. Furthermore, the `ACK` scheme follows the *end-to-end* principle which means our system functionality is enforced at the network endpoints rather than BLE and WiFi connections.

8. Conclusion

We have presented SmartSense, a system to collect and monitor data from sensors around the MIT campus. Our design prioritizes simplicity by modularizing the system and providing an intuitive communication interface. Fault tolerance and reliability are addressed by designing a scheme were sensors store and transmit a data file until receiving confirmation that the file has arrived at the central server.

To incentivize members of the MIT community, we will take measures to ensure participation is easy, such as incorporating the smartphone module into the MIT Android and iPhone apps (with user permission). Moreover, we plan to make sensor readings that require no authorized access available to the public, which appeals to the many people at MIT who are passionate about big data analysis and visualization.

We are confident that SmartSense has the potential to revolutionize ambient sensing on the MIT campus, and provide a valuable service for the Department of Facilities and members of the community alike.

Acknowledgements

The authors extend their gratitude to Prof Arvind and Manali Naik for helpful discussions.

References

[1] K. Townsend, C. Cufí, R. Davidson, Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking, Oreilly & Associates Incorporated, 2014.

[2] Power Profiles for Android (2015).
URL `https://source.android.com/devices/tech/power/index.html`