# Distributed Ambient Environment Sensing Using Mobile Devices

Amol Bhave *ambhave*
Lara Araújo *larat*
Lucas Morales *lucasem*

## 1 Introduction

With the advent of small hardware devices, it is now possible to deploy battery-powered ambient environment sensors throughout large campuses. These sensors are most commonly accompanied by expensive Bluetooth access points used to establish communication with a central server. However, purchasing and maintaining this infrastructure is financially infeasible.

In this report, we specify and analyze the design of a low-cost, reliable, durable, and configurable campus-wide ambient environment sensing system. This system is intended to be deployed on the campus of Massachusetts Institute of Technology (MIT) and to be used by MIT Department of Facilities (Facilities) and the MIT community.

To reduce the cost of this system, the expensive BLE-equipped access points are replaced by personal mobile devices to relay sensor data to the central server.

To achieve reliable communication between the sensor nodes and the central server, our design implements end-to-end acknowledgements and packet redundancy.

The ambient environment sensors consume most of their battery due to Bluetooth transmissions. To save battery and to increase the durability of our system, our design minimizes the transmission duration and frequency whenever possible. Additionally, transmissions are limited to small infrequent bursts of data packets to improve battery life.

Several aspects of our design are parametrized which makes it easy to extend with new sensors, as well as to configure the specifications of the existing ones.

The biggest achievement of our design is in its ability to provide reliable communication using inexpensive hardware.

## 2 Design Description

This design consists of three components: *sensor nodes* that record ambient environment readings and detect anomalies, the *Facilities Central Server (FCS)* which comprehensively manages sensor data and is an interface for Facilities, and the *mobile application* which uses a mobile device to act as an intermediary for communication between the FCS and the sensor nodes as well as provides a user-facing interface to the system. Additionally, the network protocol provides reliable communication between each of these components.

Sensor nodes are small *Bluetooth Low Energy (BLE)* devices deployed all over the campus to monitor the environment. Each sensor takes environment readings periodically and stores them locally. Based on its configuration, each node implements rules to proactively classify readings as *anomalous* or not. Sensor readings are transmitted using Bluetooth to nearby mobile devices and then forwarded to the FCS.

The FCS is managed by the MIT Department of Facilities. It is responsible for maintaining and reporting historical records of all sensor readings. It also provides an interface for remote configuration of sensors and alerts Facilities of any anomalous readings.

The mobile application acts as an access point for data transmission between the FCS and sensors. Additionally, it also provides an interface for members of the MIT community to query sensor data and report problems to Facilities.

The network protocol establishes communication among the three modules of our system. It uses packet replication and acknowledgements to ensure reliable transmission.

### 2.1 Sensor Nodes

A sensor node is a small BLE device used to monitor buildings on campus. These sensors may be used to

| | | | |
|---|---|---|---|
| SENSOR-ID | BUILDING-NUMBER | Major | 10 bits |
| | FLOOR-NUMBER | Major | 6 bit |
| | ROOM-NUMBER | Minor | 7 bit |
| | SENSOR-TYPE | Minor | 3 bit |
| | COLLISION-ID | Minor | 6 bit |

**Figure 1: Schema for `Sensor-ID`**

record temperature, humidity, light intensity, vibrations and detect water leaks, smoke, carbon monoxide, etc.

The sensor node includes a programmable general-purpose processor, memory (64 KB RAM and 64 KB ROM), flash storage (8MB), a real-time clock, a low-power radio for communication, and the sensor itself.

Sensor nodes collect ambient readings, detect whether they are anomalous, and transmit them to the FCS via nearby mobile devices.

#### 2.1.1 Identification

Sensor nodes are BLE devices and must have a form of identification to be used in BLE *advertisement messages*.

A BLE advertisement includes a 16-bit *major ID*, a 16-bit *minor ID* and a 128-bit *region identifier*. When a sensor node is initialized, a technician manually assigns a major and minor ID. No two sensors can have the same combination of these ID's. We define the SENSOR-ID (Figure 1) to be the concatenation of a sensor's major and minor IDs. The SENSOR-ID corresponds to the sensor location and is determined by Facilities prior to deployment.

BUILDING-NUMBER This is a 10-bit identifier assigned according to a pre-configured mapping of buildings on campus to unique numbers. Each campus has the freedom to establish the most appropriate mapping.

FLOOR-NUMBER This is a 6-bit **signed** integer corresponding to the floor in which the sensor is installed.

ROOM-NUMBER This is a 7-bit integer representing the room whose door is closest to the sensor's location.

SENSOR-TYPE This is a 3-bit identifier that maps each sensor to a type. If there are more than 7 $(2^3 - 1)$ types, the less commonly used types are assigned a special value `0` to indicate miscellaneous type.

COLLISION-ID This is a 6-bit unique identifier used to distinguish sensors that have the same location and are of the same type.

Each sensor is also configured with two system-wide shared region identifiers: `REQUEST-TO-TRANSMIT-REGION-ID` and `IDLE-REGION-ID`. See Section 2.4.1 for more details.

#### 2.1.2 Anomaly Detection

The sensor node is responsible for detecting anomalies in the environment. Each reading is classified as anomalous or not based on the sensor's configured thresholds.

Upon detecting an anomaly, the sensor attempts to communicate the event to the FCS using the network protocol described in Section 2.4.

#### 2.1.3 Local Storage

Each sensor has a 64 KB ROM, which will be used to store a 8-bit Cyclic Redundancy Check (CRC-8) calculator program and the sensor's software.

The sensor also has a 8MB flash storage, which is used to store the configuration parameters of the sensor. Therefore, in case of a power cycle, it doesn't need to be reconfigured.

In order to support fast random access, readings in a sensor node are stored in RAM with a custom data structure, referred to as an *ALL Tree*. Figure 2 and the following paragraphs describe the structure of an ALL Tree. See Section 5.1 for a sample C implementation.

**Level 1** The first level of the ALL Tree has 2 nodes which partitions the readings based on the anomalous bit.

**Level 2** In the second level, readings are organized according to the most significant 21 bits of their timestamps (in minutes). Each node in this level has at most 6 children, where each child corresponds to approximately 1.5 days' worth of data.

**Level 3** Each node in the second level points to an array of size $2^{11}$ elements indexed by the least significant 11 bits of the timestamp. Values of the array are the 16-bit readings corresponding to that timestamp.

The ALL Tree data structure provides the following operations:

ALL-GET-LATEST-READING Returns the timestamp, the value and the anomaly bit of the latest recorded reading.

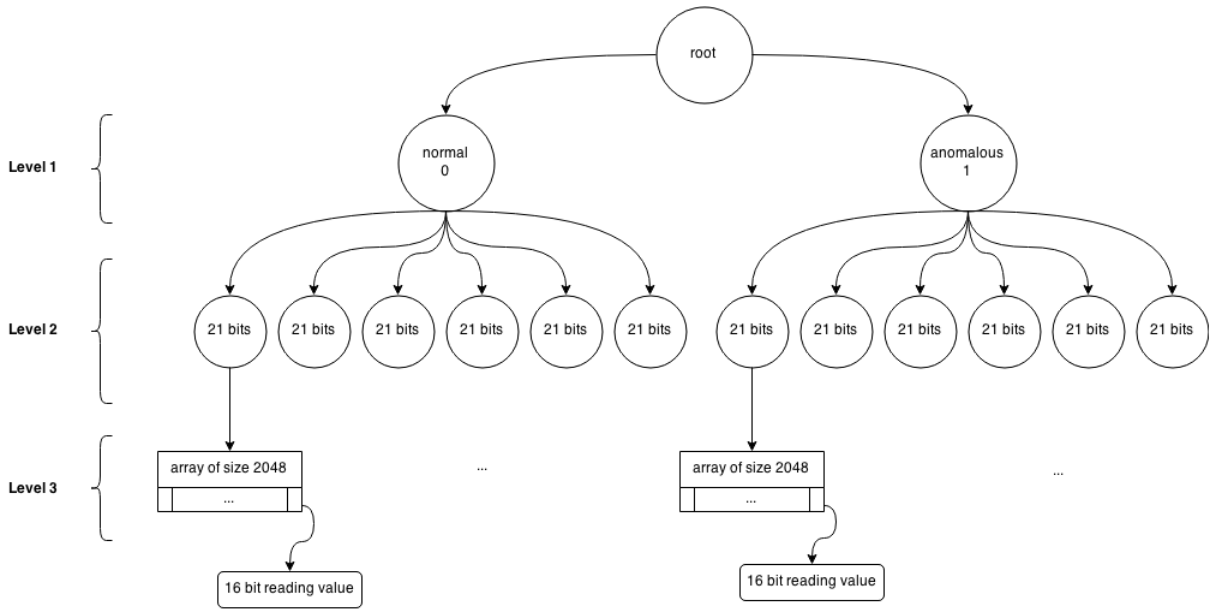


**Figure 2: Graph Representation of an ALL Tree**

ALL-DELETE-READING Takes a timestamp as input and deletes the associated reading.

ALL-APPEND-READING Appends a new reading to the ALL Tree maintaining the invariants of the data structure.

ALL-WALK Performs a in-order walk through the ALL Tree starting from the most recent to least recent anomalous reading, and then the normal readings.

In order to prevent a large backlog of readings, the ALL Tree also limits the amount of data stored in a sensor. Readings older than 7 days are relatively useless and are deleted from memory in order to increase the performance of the system. Section 3.1.3 analyses the local storage of the sensor.

### 2.1.4 Data Recording and Transmission

Each sensor records readings once every minute, broadcasts Bluetooth advertisements continuously, and connects to mobile devices periodically.

The frequency at which a sensor takes readings was decided based on the assumption that readings don't vary significantly between one-minute periods. This low recording frequency makes our readings sparsely distributed, but considerably reduces the storage complexity.

Readings are not deleted unless the sensor receives an acknowledgement from the FCS (See Section 2.4.3) or if it is beyond its storage threshold. Recording readings once every minute creates a considerably small backlog of readings in the storage. This decreases the connection time between sensor nodes and mobile devices. This also preserves the sensor nodes' battery life and thus increases the overall system performance.

Bluetooth advertisements don't consume a lot of power. However, actual Bluetooth connections have a significant power consumption. Because of this, the sensor node advertises continuously but establish connections periodically. See Section 3.3.1 for an analysis of battery-life.

As mentioned in Section 2.1.1, each sensor node is configured with two system-wide region identifiers. The sensor node chooses which region identifier to use in its advertisements depending on its willingness to transmit data. Each sensor attempts to establish one connection every 5 minutes. See Section 2.4.1 for details.

### 2.1.5 Configurability

Several parameters in the sensor nodes can be remotely configured by the FCS. Some parameters include threshold for anomalous data, frequency of transmissions, intervals for timeouts, etc.

Every time a connection to a mobile device is successful, the mobile device forwards configuration information to the sensor. Section 2.4.3 describes the communication involved in changing configurations. Upon receiv-

ing these packets, the sensor node re-configures itself and records the new parameters in its flash storage.

## 2.2 Facilities Central Server

The FCS maintains a historical record of all sensor readings, can be queried for data analysis with secure authentication for sensitive data, provides an interface for remote configuration of sensors, and reports anomalies.

### 2.2.1 Sensor Reading Database

The FCS stores all received sensor readings in a SQL database named `Records-Table`. Figure 3 describe the schema for this table. The records table has columns for the sensor READINGs, a TIMESTAMP (minutes since UNIX epoch), a hierarchical SENSOR-ID (BUILDING-NUMBER → FLOOR-NUMBER → ROOM-NUMBER → SENSOR-TYPE → COLLISION-ID, all concatenated), and a IS-ANOMALY bit. The table is indexed by TIMESTAMP, SENSOR-ID, and IS-ANOMALY to allow fast searches on these fields.

The TIMESTAMP and the SENSOR-ID together form the primary key for rows in the database.

| | |
|---|---|
| TIMESTAMP | 32 bits |
| SENSOR-ID | 32 bits |
| READING | 32 bits |
| IS-ANOMALY | 1 bit |

**Figure 3: Schema for `Records-Table`**

### 2.2.2 Receiving Sensor Readings

The FCS provides an HTTP endpoint at **/report** which receives sensor readings from mobile applications. It takes POST parameters `sensor_id`, `timestamp`, `reading`, and `is_anomaly`. It then executes the following SQL query on `Records-Table` using escaped parameters:

```
INSERT INTO Records-Table
VALUES (timestamp, sensor_id, reading, is_anomaly);
```

Additionally, the FCS keeps a `Report-Log` which is a log of received readings and the IP address of the router that the mobile device was connected to, obtained via `traceroute` or similar methods.

### 2.2.3 Handling Queries

The FCS handles queries for records via an HTTP endpoint at **/query** that uses MIT certificates for authentication. This endpoint listens for GET requests with the parameter `q` for a SQL query.

We allow various subset of queries to be performed on `Records-Table` based on the authentication of a particular user. For data analysis and data aggregation, we use the functions already provided by SQL such as `SUM`, `AVG`, and `COUNT`. To ensure that our response is limited, pagination (using SQL `LIMIT` clause) is used to only allow a small constant number (e.g. 25) of returned results.

### 2.2.4 Sensor node Information

The FCS is responsible for storing information regarding sensor nodes. The technician that installs a sensor node must store the ROUTER-IP of the closest router to that sensor node and an AFS-GROUP if applicable to limit access to sensor readings based on user authentication.

This information storage is implemented within the file system in a particular directory called the *Sensor Node Directory*. In this directory, there are $2^{16}$ sub-directories named from 0X0000 to 0XFFFF corresponding to the most significant 16 bits of SENSOR-ID. Within each of these sub-directories are $2^{16}$ *sensor node information files* corresponding to the least significant 16 bits of SENSOR-ID.

Each sensor node information file is a dictionary containing possibly multiple entries for ROUTER-IP and AFS-GROUP. See Figure 4 below for an example of one of these files.

```
Router-IP: 18.9.22.69
AFS-Group: 6.033-students
AFS-Group: 6.033-staff
```

**Figure 4: Sensor Node Information File**

### 2.2.5 Remote Configuration of Sensor Nodes

The FCS hosts a secure website at the **/fcs/** endpoint for the Facilities to remotely configure sensor nodes. This website allows the Facilities to choose specific sensor nodes and request configuration changes to them. They can also use wild card selections for sensors if some configuration change needs to be made to several sensors at once. This request, appended with a unique CONFIG-SEQUENCE-NUMBER, is forwarded to the sensor using the Network Protocol (Section 2.4.3). The sequence number helps us keep track of acknowledgements from the sensors.

Each configuration message needs to be acknowledged at **/fcs/ack** endpoint. The acknowledgement contains the CONFIG-SEQUENCE-NUMBER. If no acknowledgement has been received by the FCS after a configured timeout, it re-sends that configuration message.

#### 2.2.6 Detecting Sensor Node Displacement and Malfunction

The FCS is responsible for detecting whether sensors are displaced or are malfunctioning. It runs a nightly batch job on the `Report-Log` and does the following three checks:

1. If a certain number of configuration messages are sent to a sensor and none of them have been acknowledged, the sensor is either not in its original position near it's corresponding ROUTER-IP or it is malfunctioning.

2. If a majority of readings received from a sensor did not originate from mobile devices connected through the corresponding ROUTER-IP from the Sensor Node Directory, the sensor has mostly likely been displaced.

3. If no readings have been received from some sensor for a long period of time, it is most likely malfunctioning.

Upon completion of this batch, we notify the Facilities with the SENSOR-ID's of these possibly displaced or malfunctioning sensors. We also include the IP address of the most common router IP address for a displaced sensor, so that the Facilities can locate this sensor.

#### 2.2.7 Anomaly Notification

The sensors are responsible for detecting if any parameter of the environment is anomalous. When an anomaly record is received, the reading is forwarded in an alert to a configurable set of email addresses and phone numbers for Facilities.

#### 2.2.8 Mobile Network

The FCS uses mobile devices to communicate with sensor nodes. To facilitate this communication, the FCS must be able to select a mobile device given a ROUTER-IP. (See Sections 2.4.2 and 2.4.3) The FCS receives heartbeat messages from the mobile devices and stores their IP addresses in a map from ROUTER-IP to IP addresses. After a pre-configured timeout, if the FCS hasn't heard from some IP address, then that address gets deleted from this map.

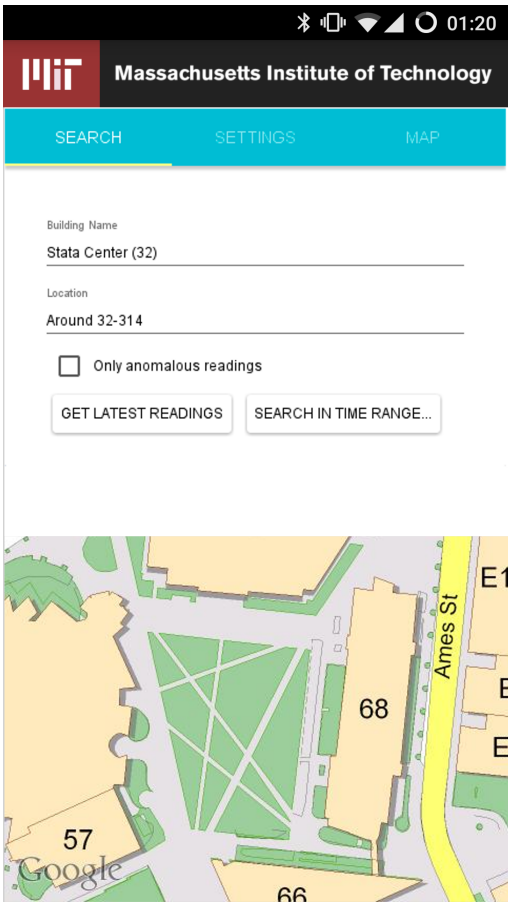


**Figure 5: Sample Mobile Application User Interface**

### 2.3 Mobile Application

MIT has an existing MIT Mobile App for Android [1] and iOS [2]. This design proposes an extension to the app due to its prevalence in the MIT community and because it already includes methods of Kerberos authentication and filing reports to Facilities.

The app allows the user to make data queries to the FCS and to manually send environmental reports to Facilities. Data analysis reports can be displayed in form of tables or graphically such as with heat-maps. The app allows the user to filter readings based on sensor location as well as a time interval, as described earlier in Section 2.2.3. Figure 5 shows a sample user interface which could be used.

The app also acts as an intermediary for communication between FCS and the sensor. The app registers two BLE region identifiers, `REQUEST-TO-TRANSMIT-REGION-ID` and `IDLE-REGION-ID`. The mobile device is configured to wake up upon receiving BLE advertisements with the `REQUEST-TO-TRANSMIT-REGION-ID` region iden-

tifier. When the mobile device receives readings or configuration acknowledgements from a sensor nodes, it is stored locally on the device until it can be sent to the FCS. The mobile device also stores configuration messages sent by the FCS to sensor nodes until it is able to successfully transmit configuration messages to their corresponding sensor node destinations. The following sections describe the details on how this communication is achieved.

## 2.4 Network Protocol

The primary purpose of the network protocol is to provide reliable communication between the FCS and the sensors over less reliable systems such as Bluetooth and the Internet. We chose reliability as our priority because we recognized that loss of sensor readings corresponds to loss of anomaly alerts, which is undesirable for this system.

The network protocol consists of two major components: the *Bluetooth Protocol* which is used for communication between the mobile device and the sensor nodes, and the *Web Protocol* which is used for communication between the FCS and the mobile device. The network protocol also specifies how the dissociated parts of the system, the FCS and the sensor, communicate with each other via an end-to-end interface implemented using these two components.

### 2.4.1 Bluetooth Protocol

The Bluetooth protocol is used for communication between the mobile device and the sensor. It is a best-effort transmission protocol and does not use acknowledgments after packet delivery. The protocol is made up of three components: the packet structure, the transmission scheme, and integrity checking.

**Packet Structure** Transmissions over Bluetooth are divided into packets. There are four types of packets, as specified in Figure 6.

`MOBILE-HELLO` This is the initial packet sent by the mobile device to the sensor when a connection is successfully established. It is used to synchronize the sensor node's clock and to send configuration updates to the sensor node. The sensor node starts sending readings to the mobile device only after receiving this packet.

`MOBILE-ACK` These packets contain FCS-administered acknowledgements of received readings from the sensor node. The mobile devices sends these packets to their corresponding sensor nodes.

`SENSOR-DATA` These packets send readings from sensor nodes to mobile devices. If the mobile application requested the latest reading, it only responds back with the latest recorded reading.

`SENSOR-CONFIG-ACK` This packet is used to acknowledge configuration messages sent by the FCS.

The 96-bit CONFIGURATION value used in the `MOBILE-HELLO` packet is simply a concatenation of CONFIG-SEQUENCE-NUMBER, CONFIG-KEY, and CONFIG-VALUE, each of which are 32 bits.

**Transmission Scheme** Bluetooth connections are always initiated by the mobile device. A sensor is preconfigured with two BLE region identifiers:

`REQUEST-TO-TRANSMIT-REGION-ID` This region identifier signals that the sensor is willing to send its stored readings and is requesting any available mobile device to connect to it. The sensor also uses this region identifier if it has detected an anomalous reading to ensure swift communication of such time-sensitive readings. After closing a connection while configured to this region ID, the sensor node will switch to using the `IDLE-REGION-ID`. Sensor nodes switches to this region ID every **5 minutes** to indicate that it wants to transmit.

`IDLE-REGION-ID` This region ID signals that a mobile device should connect to it only if it wishes to retrieve the latest sensor reading or to send a configuration message. A sensor using this region ID is not yet willing to transmit all of its readings.

The mobile device is configured to wake the application when it receives a BLE advertisement with the `REQUEST-TO-TRANSMIT-REGION-ID` region identifier. The mobile application then tries to connect to the sensor node. Upon a successful connection, it sends a `MOBILE-HELLO` packet to the sensor node.

The sensor, upon receiving the `MOBILE-HELLO` packet, does the following. It first synchronizes its clock to the timestamp received in this packet so future readings will be more accurate. If the sensor was broadcasting `REQUEST-TO-TRANSMIT-REGION-ID` and the REQUEST-LATEST-READING bit was 0, then it does a in-order traversal of the stored readings by calling ALL-WALK on the ALL Tree and sends the readings using `SENSOR-DATA` packets. However, if the REQUEST-LATEST-READING bit was 1, then the sensor gets the latest reading by calling ALL-GET-LATEST-READING and sends a single `SENSOR-DATA` packet to the mobile phone.

| | | | |
|---|---|---|---|
| MOBILE-HELLO | TIMESTAMP [32 bits] | REQUEST-LATEST-READING [1 bit] | CONFIGURATION [96 bits] |
| MOBILE-ACK | TIMESTAMP [32 bits] | | |
| SENSOR-DATA | TIMESTAMP [32 bits] | READING [16 bits] | |
| SENSOR-CONFIG-ACK | CONFIG-SEQUENCE-NUMBER [32 bits] | | |

**Figure 6: Bluetooth Packet Structure**

The mobile device may also have queued acknowledgements from the FCS for that sensor. These acknowledgements are transmitted from the mobile phone in `MOBILE-ACK` packets to the sensor and then deleted from its local storage.

The mobile device may send a configuration message in the `MOBILE-HELLO` packet. This update is acknowledged by the sensor using the `SENSOR-CONFIG-ACK` packet.

**Integrity Checking** The Bluetooth packet transmission is not reliable. It has a Bit Error Rate (BER) of 0.1% [3]. To check for transmission errors, every packet has a 9-bit Cyclic Redundancy Check-8 (CRC-8). This is calculated using a CRC-8 calculator before transmission at the source and verified at the destination. If the calculated CRC doesn't match the one in the packet, then the destination infers that the packet was corrupted and disregards it. Because sensor nodes send all of their readings when they connect, the corrupted packet will eventually be retransmitted without corruption. Thus, ignoring the lost packet does not cause a major performance impact, but instead reduces the complexity of our design.

Another aspect of this which we need to take care of is not to keep our Bluetooth channel open for too long. One way to prevent a long connection is to implement a hard configurable timeout on Bluetooth connections. Any connection which lasts longer than this timeout is closed by the sensor node. This helps preserving the battery life in case some mobile device misbehaves.

### 2.4.2 Web Protocol

The FCS and the mobile application communicate over the Internet. We use the *HTTP* protocol [4] which uses the *TCP/IP* protocol. TCP/IP is an existing protocol used on the Internet which guarantees reliable, in-order delivery of data packets. HTTP is a layer over TCP/IP which specifies a client-server model of communication between two devices. Both the FCS and the mobile device act as a client and server for the HTTP protocol.

**FCS — Server** The FCS listens on port 6144 for incoming HTTP requests from mobile devices. The FCS provides the following endpoints:

**/fcs/** This is a Web User Interface for the MIT Department of Facilities to view and analyse data, as well as to configure the sensors. This endpoint is secured by MIT certificates and is only accessible to MIT Facilities.

**/fcs/report** This endpoint receives sensor readings. It listens for a POST request with parameters `timestamp`, `sensor_id`, `reading`, and `is_anomaly`. See Section 2.2.2 for more.

**/fcs/query** This endpoint takes sensor reading queries. It listens for a GET request with an MIT Certificate and parameter `q`, a SQL query for `Records-Table`. The response is of the same format as the SQL query output. See Section 2.2.3 for more.

**/fcs/ack** This endpoint receives configuration acknowledgements from sensor nodes via mobile devices. It listens for a POST request with parameters `sensor_id`, `config_key`, `config_value`, and `timestamp`. See Section 2.2.5 for more.

**/fcs/mobile-heartbeat** This endpoint receives heartbeat messages from mobile devices. It listens for POST requests with a `router_ip` parameter. See Section 2.2.8 for more.

**Mobile Application — Client** The mobile application will relay sensor readings to the FCS. It will also query the FCS for sensor readings if the user makes a request, post configuration acknowledgements from sensor nodes, as well as send heartbeats periodically every 3 minutes. The application connects to the FCS on port 1644 using the endpoints **/fcs/report**, **/fcs/query**, and **/fcs/ack**, and **/fcs/mobile-heartbeat** to accomplish each of these tasks respectively.

**FCS — Client** The FCS also acts an HTTP client for the mobile application server. Upon receiving a sensor reading, the FCS needs to respond to the mobile application with an acknowledgement. Additionally, any configuration messages will also need to be relayed to the sensors. As will be explained in the next section, the FCS will use location-based routers as a means to find a mobile application server.

**Mobile Application — Server** The mobile application in every mobile device acts as a HTTP server and listens on port 6145. This server is meant to receive requests from the FCS about sensor data acknowledgements and configuration changes. This server provides the following endpoints:

**/mobile/ack** This endpoints listens for a POST request with parameters `sensor_id` and `timestamp`. The FCS sends a request to this endpoint to signify an acknowledgement for a sensor data reading.

**/mobile/config** This endpoints is used to make configuration changes to the sensors. It listens for POST request with parameters `sensor_id`, `key`, and `value` which signifies a key-value pair to be configure on the given sensor.

#### 2.4.3 End-to-End Data Transmission

Both the Bluetooth protocol and the Web protocol work together with mobile devices as a medium to achieve communication between the FCS and the sensor.

**Sensor to FCS** A transmission from a sensor node to the FCS works in the following way. Upon connecting to a mobile device, the sensor sends all its data readings and any configuration acknowledgements to a mobile device using the Bluetooth Protocol (Section 2.4.1). The mobile device stores these messages in its local storage until it has an Internet connection. Once the mobile is online, the mobile application uses the Web Protocol (Section 2.4.2) to send the messages to the FCS. The mobile phone then deletes these stored messages after posting the message to the FCS and receiving the appropriate HTTP response, confirming that the FCS has successfully and reliably received the message.

**FCS to Sensor** The FCS needs to transmit reading acknowledgements and configuration messages to the sensor nodes. To send these messages to a sensor node, the FCS first needs to know what mobile device to send the request to. It uses the Sensor Node Directory to get a ROUTER-IP (or many of these addresses, if the sensor node information file has multiple entries) given the SENSOR-ID. The IP address corresponds to the nearest router to a sensor node. The FCS searches for a mobile device connected through this router via the mobile application and forwards all requests to that mobile device.

The mobile device, upon receiving this request, tries to deliver it to the sensor. In order to increase the chances of delivery, we do the following. The mobile device broadcasts this request to the *broadcast IP address* of the router. All the devices connected to that routers will respond back and the request will be forwarded to all of these devices. Then, if any mobile device among those happen to connect to the sensor, that device will forward the transmission to it.

If the request was for a configuration update, the sensor acknowledges back a `SENSOR-CONFIG-ACK` packet which gets forwarded to the FCS using the previously described route.

## 3 Analysis

### 3.1 Scalability

#### 3.1.1 Campus size

Our choice of encoding the location of a sensor in the major and minor ID limits the size of the campus our system can be deployed to.

As described in Section 2.1.1, our system can support up to 1024 distinct buildings, 64 floor levels and 128 room numbers. Further, it can support 512 sensor nodes, or up to 64 sensors nodes of the same type, around any single location, to the nearest room. Therefore, even though limited, our system can still be used in a very large campus.

#### 3.1.2 Adding New Sensors

Adding a new sensor to our system is very simple. We only need to deploy the sensor, configure it with a new SENSOR-ID and add this entry to the Sensor Node Directory. The new sensor will then start recording data immediately and become a part of our system.

#### 3.1.3 Sensor Node Storage

To improve performance and to provide fast random access to readings, the system store sensor readings in a special data structure called the ALL Tree that resides in the sensor's 64KB RAM.

Because we record readings every minute, the readings' timestamps are stored as minutes since *UNIX epoch* [5], so the lower 11 bits of the timestamp correspond to $2^{11} = 2048$ min $= 1.42$ days worth of data. Because we

disregard data older than 7 days, the second level of the stored ALL Tree will have at most $\frac{7}{1.42} = 4.6 < 6$ nodes (see Figure 2). That way, the maximum amount of data used for storing readings in the ALL Tree is

$$2 \cdot 6 \cdot 2048 \cdot 16 = 393216 \text{ bits} = 48 \text{ KB},$$

which is less than the size of sensor's RAM.

Since we have very limited memory after allocating the ALL Tree, changing certain parameters in the system may mean that we can no longer store the readings in the memory. For example, changing the size of the readings to a larger value, increasing the frequency with which readings are taken, and so on may lead to problems with this in-memory storage.

However, an in-memory storage provides the advantage of fast random access to sensor readings indexed by timestamp which considerably increases the performance of our system by decreasing the connection time between sensors and mobile devices.

#### 3.1.4 Large Influx of Users

The choice to have two different region identifiers, allows our design to behave well even with a large number of mobile devices in the vicinity of a sensor.

A mobile phone connects to a sensor if it sees a `REQUEST-TO-TRANSMIT-REGION-ID` region identifier. The sensor only sets this region identifier if it wants to connect. Therefore, the number of connections made is controlled by the sensor as opposed to the mobile devices and thus, the connection time is not affected by the number of nearby mobile devices.

#### 3.1.5 Mobile Device Intermediary

One of the main characteristics of our design is replacing expensive Bluetooth access points by personal mobile devices. This greatly decreases the cost of our system and makes its deployment financially feasible, while still maintaining a reliable data transmission between the sensors and the FCS.

### 3.2 Reliability

#### 3.2.1 Loss Rate of Sensor Readings

The communication between sensor nodes and the FCS is designed to minimize data loss.

Ideally, a mobile device is always connected to the Internet. However, it is possible for a mobile device to store sensor data and to never transmit it to the FCS because it was never able to connect to the Internet.

Due to the system of acknowledgements, even if some data packet failed to deliver, it will be retransmitted by the sensor node to some other device.

A data packet is transmitted to multiple devices, which gets stored in that device until the packet expires. So, in order to transmit the packet, we only need one of these devices to be connected to the Internet before the packet expires. This greatly increases our chances of delivering a packet to the FCS.

This use of end-to-end acknowledgements and packet redundancy greatly increases the reliability of the system so, the chances of an actual data loss occurring is very low.

#### 3.2.2 Anomaly Report

Anomaly report is a key feature of our system and our design attempts to alert about these events as soon as possible. Upon detecting anomalies, the sensor immediately switches to using `REQUEST-TO-TRANSMIT-REGION-ID`. This signals nearby mobile devices to connect to the sensor if possible and get sensor readings. Therefore, ideally an anomalous reading should be immediately transmitted to the FCS. However, we cannot assume that every mobile device is connected to the Internet.

Assuming that 50% of the devices are not connected to the Internet, the expected number of times you need to connected to a mobile device in order to reach a device that is connected to the Internet is 2. Since the interval between connections is 5 minutes, the expected latency time for an anomalous reading is 5 minutes.

### 3.3 Durability

#### 3.3.1 Sensor Node Battery Life

Throughout our design, decisions such as the frequency and timeout for connections were made with the sensor's battery life in mind. We optimized several parameters of our system in order to achieve reliable communication while allowing sensor's to last more than 4 years.

We will assume that the probability of a sent sensor reading being acknowledged back to the sensor is 1%. This is estimate is used to find out the expected number of re-transmissions required per data packet.

The 1% probability of acknowledgement implies that the expected number of times we need to resend a sensor reading until it finally gets acknowledged is 100.

The sensor record new readings once per minute and connect at most once every 5 minutes. Therefore, the expected number of readings stored in a sensor will be $1 \cdot 5 \cdot 100 = 500$.

Since each reading corresponds to a 74-bit data packet and in every connection the sensor attempts to send all its data, the expected amount of data sent per connection is 74$\cdot$ 500 = 37000 bits = 4.52 kbytes.

Considering the 4kbytes/s BLE transmission rate and the 2 additional seconds it takes to establish a connection,

the expected duration of each sensor connection will be $2 + \frac{4.52}{4} = 3.13$ seconds.

The sensor nodes draws 1mA during data transmission. So, in 4 years, having a connection interval of 5 minutes, will perform at most $\frac{4 \cdot 365 \cdot 24 \cdot 60}{5} = 420480$ connections. So, the expected battery usage for transmission will be $1 \cdot 420480 \cdot 3.13 = 1316102.4$ mA.s $= 365.58$ mA.h.

The sensor also consumes energy to broadcast advertisements. Broadcasting at 1Hz, the average current draw is 0.025mA over an entire second. Since we advertise continuously, the total battery usage in a 4 year period is $0.025 \cdot 4 \cdot 365 \cdot 24 = 876$ mA.h.

Therefore, over a period of 4 years, the expected battery usage of a sensor node is $876 + 365.58 = 1241.58$. Since each sensor node has a 1600mA.h battery, the average lifetime of our sensor is $\frac{1600}{1241.58} \cdot 4 = 5.15$ years.

#### 3.3.2 Handling Malicious Behavior

Our system is capable of defending itself against some types of malicious behavior.

For example, if a mobile device misbehaves and slows down a Bluetooth connection with a sensor, this may cause a huge batter drain in the sensor. However, our design protects against such actions by doing the following: the sensor node will still connect to a faulty device, but it will only remain connected for a limited period of time, as described in Section 2.4.1. By default the Bluetooth connection timeout is set to be 5 seconds. By performing calculations similar to the ones in the above section, we came to the conclusion that such timeout asserts a battery life of at least 4.38 years.

#### 3.3.3 Effects of hardware upgrades

Hardware upgrades will most of the times increase the performance of our system. For example, faster Bluetooth connections will decrease the connection time, decreasing latency and saving battery life. However, if a sensor increases its readings precision and starts storing 32-bit reading values, this might jeopardized the local storage of the sensor. Yet, if this is accompanied by an increase in the size of the sensor's RAM, our system will remain unaffected.

A drastic hardware upgrade such as MIT not using routers through out its campus anymore might render our system unusable, but we assume that this is not very likely to happen in the nearby future.

### 3.4 Configurability

One of the requirements on our system is that it must be configurable. Any changes in the parameters of the sensors must be easy to perform. Our design makes it possible to configure several parameters of the sensor. The MIT Facilities staff can login to the web interface and specify the configuration changes they wish to make. Our network protocol is then used to transmit these changes to the sensor.

### 3.5 Fault Tolerance

#### 3.5.1 Hardware Failures

There are several ways the hardware could fail and our system needs to be able to tolerate it. It is possible for our sensor node to undergo a power cycle. This is a problem for our system because we store the readings in volatile memory and hence are erased upon restart. However, since we store the configuration parameters in the flash storage, the sensor could use these saved parameters and become functional immediately.

Our system also is capable of addressing sensor node failure and displacement (see Section 2.2.6).

#### 3.5.2 Network Failures

Network failures can happen frequently and our design is able to handle them properly. We need to mainly deal with two types of failures: packet corruption and loss. To mitigate these, we make use of TCP/IP which itself guarantees reliability and CRC-8 checksum in the Bluetooth transmission to check for corrupt packets.

Our system is also able to deal with short network shortages. If the Wi-Fi does down in a certain location on campus, mobile phones in that vicinity won't be able to communicate with the FCS. However, they will still gather sensor data and retain it in memory until it timeouts. That way, if the device moves to a region with Wi-Fi coverage, it will transmit data from sensors that are not in Wi-Fi zone.

### 3.6 Use Cases

#### 3.6.1 Archiving

The database in the FCS is an archive all the sensor readings. Each sensor reading from all the sensor is stored in the FCS and can be queried and analyzed anytime from the mobile application or from the FCS web interface available for the MIT Facilities.

#### 3.6.2 Anomaly detection

Our system can detect anomalous sensor data by using simple rules such as a threshold for the sensor reading. If the sensor flag a reading as anomalous, it tries its best to send to notify the FCS about it. See Section 3.2.2 for more details.

### 3.6.3 User reports of issues

The MIT Mobile app has the basic infrastructure for contacting Facilities to report on-campus maintenance task. We augment this infrastructure with additional fields relevant to our system.

Users can use the app to report problems with the sensor, as well as can notify the Facilities themselves if there is a maintenance issue such as leaks or lighting.

### 3.6.4 User retrieval of archived information

Each sensor is mapped to an AFS-GROUP-IDENTIFIER. A user can use the mobile application to query and analyze the sensor data. Only a user who is a member of an AFS Group with this identifier will have access to the readings for this sensor. See Section 2.3 for more details.

# 4 Conclusion

This report proposes a design for a low-cost, reliable, durable, and configurable campus-wide ambient environment sensing system. The system provides an interface with the mobile application, allowing users to query sensor information for any MIT building, floor, or room. Environmental anomalies are also detected and reported to Facilities to resolve the issue.

We started by replacing the expensive parts of the most common environment sensing solution with personal mobile devices, which introduced additional challenges to communication that lead to the construction of a reliable network protocol for communication between the modules of our system. We designed our system to achieve all the important tasks any environment sensing system needs: sensing and storing ambient readings, and notifying relevant parties of any anomalous readings in the environment.

In this way, our design makes it feasible to deploy a campus wide ambient environment sensing system using mobile devices.

# 5 Appendix

## 5.1 ALL Tree

For storing readings in sensors, this is the implementation our data structure, the ALL Tree.

```
#include<stdlib.h>
#include<stdint.h>
#include<time.h>

extern void ALL_walk_func(ALL_reading reading);

typedef struct {
    ALL_node* anom;
    ALL_node* nonanom;
} ALL_tree;

typedef struct {
    int start_offset; // reading[0] shares these first 21 bits of timestamp
    uint16_t* readings[6]; // about 8 days, where readings[i] is an array of size 2^11 (4KB)
} ALL_node;

typedef struct {
    bool anomalous;
    int timestamp;
    uint16_t reading;
} ALL_reading;

int minute_time() {
    time_t clk = time(NULL);
    return ((int)clk) / 60;
}

uint16_t* new_readings() {
    // array of size 2^11 filled with 1 bits (-1 values).
    return (uint16_t)memset(malloc(1<<11, sizeof(uint16_t)), -1, 1<<11);
}

ALL_tree* ALL_initialize() {
    ALL_tree* root;
    ALL_node* anomalous, nonanomalous;
    root = (ALL_tree*)malloc(sizeof(ALL_tree));
    anom = (ALL_node*)malloc(sizeof(ALL_node));
    nonanom = (ALL_node*)malloc(sizeof(ALL_node));
    root->anom = anom;
    root->nonanom = nonanom;
    int time = minute_time();
    anom->start_offset = time;
    nonanom->start_offset = time;
}

int ALL_append_reading(ALL_tree* root, ALL_reading item) {
    int upper,lower;
    ALL_node* node;
    uint16_t* readings;
    node = item->anomalous? root->anom : root->nonanom;
    upper = item->timestamp - node->start_offset >> (32-21);
    lower = item->timestamp & ((1<<12)-1);
    if (upper < 0) return 1;
    while (upper > 6) {
        free(node->readings[0]);
        node->readings[0] = node->readings[1];
        node->readings[1] = node->readings[2];
        node->readings[2] = node->readings[3];
        node->readings[3] = node->readings[4];
        node->readings[4] = node->readings[5];
        node->readings[5] = new_readings();
        upper--;
    }
    readings = node->readings[upper];
    if (!readings) node->readings[upper] = new_readings();
    readings[lower] = item->reading;
    return 0;
}

int ALL_delete_reading(ALL_tree* root, int timestamp) {
    ALL_node* node;
    int upper,lower;
    uint16_t* readings;
    node = root->nonanom;
    for (a=0; a<2; a++) { // once for anom, once for nonanom
        upper = timestamp - node->start_offset >> (32-21);
        lower = timestamp & ((1<<12)-1);
        if (upper < 0 || upper > 5) return -1;
        readings = node->readings[upper];
        if (!readings) continue;
```

```
81          if (readings[lower] != -1) {
82              readings[lower] = -1;
83              return 0; // deletes nonanom before
                    anom if timestamp collision
84          }
85          node = root->anom;
86      }
87      return 1;
88  }
89
90  ALL_reading ALL_get_latest_readings(ALL_tree* root
        ) {
91      ALL_node* node;
92      uint16_t* readings;
93      uint16_t reading;
94      int timestamp;
95      node = root->anom;
96      int a;
97      for (a=0; a<2; a++) { // once for anom, once
            for nonanom
98          timestamp = node->start_offset;
99          int u;
100         for (u=5; u>=0; u--) {
101             readings = node->readings[i];
102             if (!readings) continue;
103             for (l=(1<<11)-1; l>=0; l--) {
104                 reading = *(readings++);
105                 if (reading != -1) return (
                        ALL_reading){
106                     .anomalous = !a,
107                     .timestamp = timestamp+l,
108                     .reading   = reading
109                 };
110             }
111             timestamp += 1<<12;
112         }
113         node = root->nonanom;
114     }
115 }
116
117 void ALL_walk(ALL_tree* root) {
118     ALL_node* node;
119     uint16_t* readings;
120     uint16_t reading;
121     int timestamp;
122     node = root->anom;
123     int a;
124     for (a=0; a<2; a++) { // once for anom, once
            for nonanom
125         timestamp = node->start_offset;
126         int u;
127         for (u=5; u>=0; u--) {
128             readings = node->readings[i];
129             if (!readings) continue;
130             for (l=(1<<11)-1; l>=0; l--) {
131                 reading = *(readings++);
132                 if (reading != -1) {
133                     ALL_walk_func((ALL_reading){
134                         .anomalous = !a,
135                         .timestamp = timestamp+l,
136                         .reading   = reading
137                     });
138                 }
139             }
140             timestamp += 1<<12;
141         }
142         node = root->nonanom;
143     }
144 }
```

# 6 Glossary

**Advertisement Messages** Periodic messages sent by a BLE device which lets other BLE devices connect to it.

**ALL Tree** An efficient data structure for the storage of readings within a sensor node. ALL is made of the initial letters of the given names of the authors of this paper. See Section 2.1.3 for details, and Section 5.1 for the implementation.

**Anomaly** A reading taken by a *sensor node* which, based on a certain threshold, is considered to be abnormal.

**BLE** A communication technology which uses a low-power radio with an energy-efficient link-layer protocol and a low communication range.

**Bluetooth Low Energy** See *BLE*

**Broadcast IP Address** An IP address which exists on a local network on which if a packet is sent, it gets broadcasted to all the devices connected in that local network.

**Facilities Central Server** See *FCS*

**FCS** A central server managed by the MIT Department of Facilities used to store and query historical sensor readings and to notify facilities in case of anomalies.

**HTTP** Also Hypertext Transfer Protocol. A transport protocol used on the Internet using a client-server model.

**Major ID** A 16-bit field in a BLE advertisement.

**Minor ID** A 16-bit field in a BLE advertisement.

**Mobile Application** An extension proposed to the MIT Mobile App to support the ambient sensing system.

**Region Identifier** A 128-bit field in a BLE advertisement.

**Sensor Node** small size *BLE* devices deployed all over campus used to monitor the ambient environment.

**Sensor Node Directory** A map of SENSOR-ID to information about the sensor.

**Sensor Node Information File** A file for an individual sensor node stored on the FCS containing at least one of the sensor node's nearest routers' IP addresses, as well as any AFS-Groups that correspond to necessary authorization to view readings for the sensor node.

**TCP** Also Transmission Control Protocol. A transport protocol used on the Internet which guarantees reliability and in-order delivery of packets.

**Unix Epoch** The date-time midnight Thursday, 1 January 1970. A timestamp is measured as number of seconds elapsed since this date-time.

**UUID** The unique universal identifier is an identifier standard in software construction. It's a 128-bit value and is deemed "practically unique".

## References


[1] MIT Mobile Android Application, Google Play Store
`https://play.google.com/store/apps/details?id=edu.mit.mitmobile2`

[2] MIT Mobile iOS Application, Apple iTunes
`https://itunes.apple.com/us/app/mit-mobile/id353590319`

[3] Bluetooth: Radio Architecture. *Bluetooth SIG, Inc.*
`https://developer.bluetooth.org/TechnologyOverview/Pages/Radio.aspx`

[4] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. *Network Working Group*.
`http://tools.ietf.org/html/rfc2616`

[5] Unix time
`https://en.wikipedia.org/wiki/Unix_time`