Data Aggregation & Analysis System for Intelligent Agriculture: Design Document

CSci 5221 Team #1: Panagiotis Stanitsas [*1], Jacob Quant [†1] and Nabil Cheikh [‡1]

[1]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA

1 Motivation

Over the past decade a great amount of attention has been received by the area of intelligent systems development in Precision Agriculture. Methods along the lines of satellite and drone image processing as well as technologically advanced sensing tare deployed in an effort to increase the efficiency of modern crops while optimizing the quantities of the necessary resources (e.g. fertilizers, water).

Figure 1: A High-Level Overview of Precision Agriculture.

Figure 1 illustrates a high-level view of the the multi-stage structure of modern Precision Agriculture according to which information is aggregated at a central hub, then processed and based on the results, all agents receive a plan of action regarding irrigation, fertilization, harvesting and planting. Capitalizing on the continuous effort of reducing the associated cost with Precision Agriculture applications, cheap sensing solutions can be implemented in order to provide valuable information even on the farmer's smart phones. In particular, the scenario that this project aims to replicate is the delivery of information, which is harvested in the field, to a mobile device based on user defined queries. Sensors that can measure the humidity of the soil can be purchased at a very low price ($10 - $20 per sensor), while constructing a wireless network (e.g. ZigBee modules) that is capable of transmitting the sensor measurements to a central processing location can be also completed within a constricted budget. Figure 2 presents a soil humidity sensor distributed by Hoskin Scientific and has the ability to measure water content and temperature directly within the root zone. Rather than using sophisticated devices to deliver information to the farmer, this effort promotes the use of a smart phone application, while remaining aligned with the initial goal of keeping the cost as low as possible. In that way, utilizing low-cost sensor measurements, farmers can monitor the humidity (of the soil) and plan accordingly for their irrigation strategies.

[*]stani078@umn.edu
[†]quant006@umn.edu
[‡]cheik004@d.umn.edu

2 Design Overview

In summary, our team intends to develop an Android application, capable of delivering heat-map visualizations of humidity and temperature (if time allows) time series for user defined locations and time periods.

Figure 2: Soil humidity sensor

The proposed methodology has three discrete components. The first, is concerned with the sensors' deployment as well as the collection of measurements in the field. For the purposes of this study data will be simulated using appropriately selected random number generators since the time-frame does not allow experimentation with those hardware components as well. The second component includes multiple modules and refers to tasks that the server needs to perform. Three modules are associated with the server's component: a collection and storage module, a data analysis and visualization module, and a module to process requests received from the end-user using these other modules. The last component is the client application, which runs on an Android device and is responsible for submitting the user's queries and displaying the results. Figure 3 illustrates the relationships among the different components of this system. The sections that follow go into more detail about how each part of this system accomplishes its function.

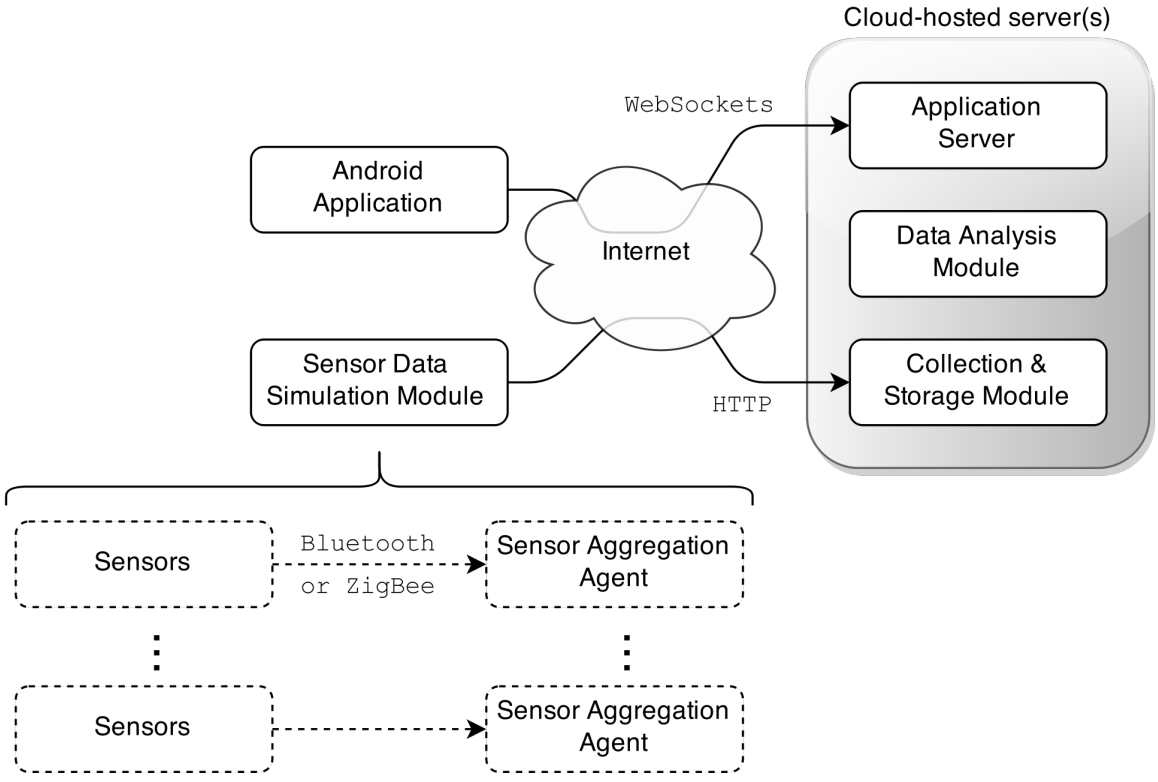


Figure 3: System overview

3 Android Application

When the Android application starts it displays a list of known servers. This list may be initially empty or pre-populated, but the user can add or remove servers using the interface presented on this screen, and these changes will be stored on the device. The application will attempt to contact each server and indicate whether or not it returned a successful response (e.g. the entry may be colored green or red). This is illustrated in Figure 4(a).

Once the user selects a server, the application establishes a Websockets connection with the server and displays a similar menu that lists the "regions" available on that server. This is illustrated in Figure 4(b). (For faster perceived performance the application actually requests this information when it first contacts the server to see if it is responding rather than waiting for the user to select it.) Regions are simply logical divisions of space being monitored by sensors. For example, a region may correspond to a field of crops, a green house, or the lawn. For simplicity, we constrain all regions to be rectangular, but this is only a user-interface limitation; the sensors do not need to be placed in any particular pattern nor does the actual field need to be rectangular (though it typically is).

After selecting a region, the user encounters the "region dashboard" screen, shown in Figure 4(c), wherein they can choose to view an automatically updating chart showing recent measurement data, request a heat map from the server, or possibly perform other tasks.

If the user opts to view the real-time chart, the app uses its Websockets connection with the server to continually obtain the latest available information and update the chart widgets as needed. The data presented by charts shown on this screen may correspond to individual sensors or statistics based on data from multiple sensors (e.g. the average moisture content over the entire region).

If the user opts to request a heat map they are presented with another interface like the one shown in Figure 4(e), which allows a rectangular "subregion" of the selected region to be chosen as well as a time frame for the heat map. After submitting the request to the server, the application waits for a response and then displays it via the results view shown in Figure 4(f) (or times out and presents an error message). On that screen the user can use a slider control to view the heat map for different times over the requested interval. The user may also be able to save the results for later reference.

4 Application Server

The application server uses Tornado[1], a Python web framework and asynchronous networking library that supports Websockets[2] to communicate with the Android application. When a client establishes a connection, it will typically begin by sending a request for the regions available on this server. This type of message may simply look like this:

```
{
    type: "getRegionList",
}
```

The server will then query the collection and storage module to obtain this information and respond with a list of regions (name and geometry) and the types of measurements taken there. For example:

```
{
    type: "regionList",
    regions: [
        {
```

[1] `http://www.tornadoweb.org/en/stable/`

[2] That is, RFC6455: `https://tools.ietf.org/html/rfc6455`

(a) Home / server menu (b) Region menu (c) Region dashboard

(d) Real-time chart (e) Heat map request builder (f) Heat map result viewer

Figure 4: User interface of Android application

```
            id: 1,
            name: "Soybean field #1",
            channels: ["moisture"],
            type: "rectangle",
            parameters: {
                nw: {
                    x: 0,
                    y: 0
                },
                se: {
                    x: 10,
                    y: 10
                }
            }
        },
    ]
}
```

If there is a problem with a request or if an error is encountered the server may alternatively respond with an error message such as the one shown below. This is a possible response to any of the client's requests.

```
{
    type: "error",
    message: "Could not retrieve region list from collection & storage module"
}
```

If the Android application user wants to view the live charts the application will send a message of the following form:

```
{
    type: "startStreamingData",
    averages: ["moisture"]
}
```

If a subsequent `startStreamingData` request is received from the application it supercedes the current one. Also, when the user navigates away from the screen showing the chart(s) the application will send a stop message, such as the one show below. Obviously, the server will also stop streaming data to the client if the connection is terminated.

```
{
    type: "stopStreamingData"
}
```

If the user requests a heat map the Android application will send a message like the following:

```
{
    type: "getHeatMap",
    channels: ["moisture"],
    region: 1,
    subregion: {
        type: "rectangle",
        nw: {
            x: 1,
```

```
            y: 1
        },
        se: {
            x: 3,
            y: 3
        }
    },
    timeSpan: {
        start: "2015-03-01T06:58:17.922Z",
        end: "2015-03-01T23:00:00.000Z"
    }
}
```

The application server will validate the request (e.g. ensure that subregion is contained in region) then see if the result is already available (e.g. from a previous request). If the results are already available they will be sent immediately. Otherwise the application server will send an equivalent request to the data analysis module, which will compute the results, and then cache and return those to the client.

5 Data Analysis Module

The Data Analysis Module (DAM) will be written in Python and will be responsible for analyzing the data that correspond to the user's request. The **input** to this module is a 3D array of dimensions **Nx3xT** where **N** is the number of sensors in the user defined region of interest and **T** is the number of time frames that the user requested. For each sensor and each time-frame its x-y coordinates along with the associated measurements are provided.

Two blocks can be identified in the intended implementation of this module. The first block is responsible for solving a 2D interpolation problem for which the locations of a small number of point measurements are available and an estimate about the humidity conditions across a larger grid of points needs to be computed.

During this preliminary investigation phase, the **Kriging** method for interpolating spatial data was tested; the computational complexity of the **Kriging** method makes it a bad candidate for the targeted implementation. Even though it is a very accurate method for this task, a simpler implementation using multiple 2D Gaussian distributions centered at the locations that measurements are available is more appropriate. Figure 5(a) presents 5 Gaussian distributions placed on a 2D grid emulating the aforementioned scenario, thus the case of having 5 sensors. Figure 5 (b) presents an existing implementation that superimposes the computed heat map on GoogleMaps for better visualization. If time allows this feature will also become available in our implementation.

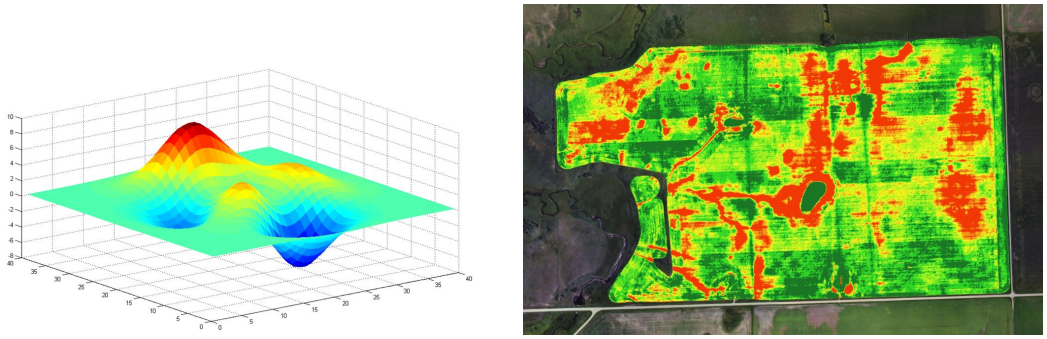
The second block of this module is responsible for harvesting the **T** heat maps generated in the above implementation and returning a video object. The implementation of this task will be performed using the **Python Imaging Libray (PIL)**[3] and the output will be the user defined video object. Further details of how this module will communicate with the others will be defined, as needed, once these modules are more mature.

6 Collection & Storage Module

The collection and storage module will use MongoDB[4], a "NoSQL" database, to record sensor and measurement information. Since MongoDB does not require the data it stores to be structured in

[3] http://www.pythonware.com/products/pil/

[4] http://www.mongodb.org/



(a) (b)

Figure 5: (a) 5 Gaussian distributions placed on a 2D grid. (b) Superimposing the heat map on the GoogleMaps.

a particular way, no specific schema needs to be defined. However, for planning purposes, we have loosely described the information that will be stored in Figure 6.

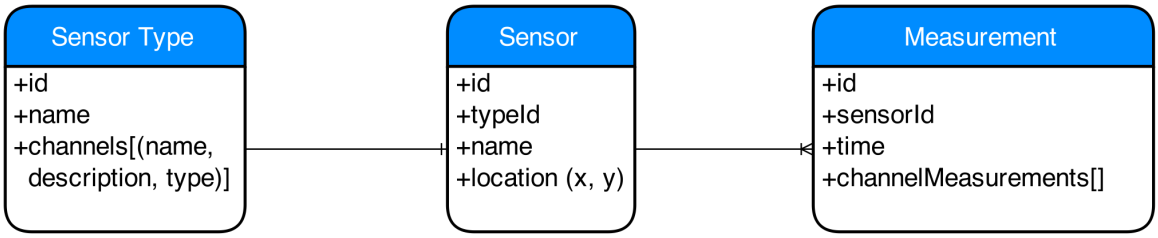


Figure 6: "Soft" database schema

This module may also store results of computations from the data analysis module, but we are deferring the decision of how this should be done because it will depend on the output of the data analysis module. Also, it may provide its own HTTP server to receive measurement submissions or it may share the Tornado instance used by the application server.

7 Sensor Data Simulator

In lieu of physical sensors collecting "real world" measurements, this module simulates a collection of virtual sensors. For each dimension (e.g. moisture content, temperature, etc.), rather than simply generating random, independent values for each sensor at each sample time we want to generate more realistic data. To do this, we define a model that takes into consideration both the time of the measurement (e.g. daily temperature cycles and seasonal variation) and the location of the sensor (e.g. perhaps some portion of the region being modeled has a lower elevation or less permeable soil). Perhaps the most important aspect of this component is not so much that it generate meaningful data but rather that it be replaceable by a system of real sensors without any disruption to the downstream components that process its output. An example of what the body of a message sent to the collection & storage module from here might look.

```
{
    definitions: {
        types: [{
            id: 1,
```

```
            name: "Temperature & moisture sensing probe",
            channels: [{
                name: "t",
                description: "Temperature in hundredths of degrees Celcius",
                type: "integer"
            }, {
                name: "m",
                description: "Moisture content in hundredths of percentage points",
                type: "integer"
            }]
        }],
        devices: [
            {
                id: 1,
                type: 1
                name: "Probe 1",
                location: {
                    x: 0,
                    y: 0
                }
            }, {
                id: 2,
                type: 1
                name: "Probe 2",
                location: {
                    x: 1,
                    y: 0
                }
            } // ...
        ]
    }
    measurements: [
        {
            id: 1,
            data: [
                {
                    time: "2015-02-11T18:59:29.729240",
                    t: 2200,
                    m: 2000
                }, {
                    time: "2015-02-11T19:00:03.659181",
                    t: 2201,
                    m: 2000
                } // ...
            ]
        }, {
            id: 2,
            data: [/* ... */]
        } // ...
    ]
}
```