MOBILE AIR QUALITY MONITORING AND WEB-BASED VISUALIZATION

By

Ronald William Hedgecock II

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December, 2009

Nashville, Tennessee

Approved:

Dr. Akos Ledeczi

Dr. Xenofon D. Koutsoukos

ACKNOWLEDGMENTS

This work was made possible by a generous grant from Microsoft, NSF award 0807464, and the continuing financial support of Vanderbilt University. Also thanks to Dr. Akos Ledeczi for mentoring and advising me throughout this process.

TABLE OF CONTENTS

Appendix

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In 2007, a collaborative research project was commissioned by Microsoft to design a system capable of providing real-time air quality information to the general public via its SensorMap online visualization interface. Work was begun on such a system with the initial steps toward its implementation being completed by late 2007 [36]. The reason behind the need for such a system comes from a serious lack of real-time pollution indicators in both the United States and worldwide. Air pollution is one of the leading causes of respiratory health problems, and with over 50% of the world's population now considered urban [20], it is up to cities to take the initiative to decrease air pollution.

The current method of air pollution monitoring in the United States includes sampling airborne pollutants on an hourly basis, averaging these pollutant concentrations together over a 24-hour period, and then publishing this data as a single value known as the day's Air Quality Index (AQI) [10]. For people who desire a higher level of resolution regarding air pollution, the Environmental Protection Agency (EPA), along with several other government agencies, runs a service called AIRNow which publishes the available hourly pollution data on a web site in both graphical and downloadable form [1]. It should be noted, however, that there are only about 5,000 pollution monitoring stations located throughout the entire country. This minimal number of sensors coupled with a sparse sensing schedule means that the AIRNow interface still only provides an extremely low-resolution image of air quality throughout the country. Since pollution is highly location dependent, there is insufficient data to accurately evaluate air quality within specific neighborhoods or at well-defined precise locations.

Implementing a mobile air quality monitoring network enables a detailed picture of air pollution to be constructed based on real-time data from mobile sensors over an entire populated area. The work carried out under the Microsoft Research Grant enabled us to demonstrate the feasibility of this approach and even build five car-mounted air pollution

sensor prototypes, each with an onboard GPS receiver and three gas sensors measuring O_3, NO_2, and CO [36]. The research outlined in this thesis consists of improving upon the original prototype sensor design, devising a method for actual implementation of a monitoring system consisting of a number of vehicle-mounted sensor nodes, evaluating advanced data processing algorithms to reconstruct an air pollution model from irregularly sampled spatiotemporal observations, and creating a set of innovative web-based applications for visualizing the data in an intuitive and easy-to-use interface while providing several additional pollution and health-related services to the public.

CHAPTER II

HARDWARE ARCHITECTURE AND FIRMWARE

Sensor Node Description

In order to discuss current research and improvements to the system, we must first introduce the building blocks of the entire mobile monitoring network, namely the vehicle-mounted sensing devices, which we will call ”sensor nodes” from here on, or simply ”nodes” for short. Figure 1 shows an actual picture of one of the prototype nodes built in 2007.

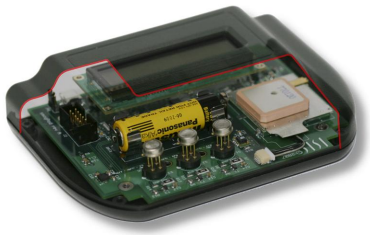


Figure 1: Sensor Node Prototype

Each of these nodes is able to collect pollution data corresponding to atmospheric O_3, NO_2, and CO levels in an autonomous fashion, as well as store this data offline for later retrieval or stream the data to a base station in real time. Data is transmitted via either an integrated Bluetooth module for wireless devices, such as laptops or PDAs, or via a built-in USB interface for wired connections. The USB interface also provides the means for powering the device, including charging its integrated Lithium ion battery. Although the battery life of the device is limited to a few hours, the nodes will be mounted on vehicles which can provide power while running, so this is not a limiting factor in practical applications. Furthermore, each node contains a 2-axis MEMS accelerometer to detect whether the node is in motion and turn off power-hungry components such as GPS, Bluetooth, and the digital LCD if stopped. Location and time information is provided by an on-board 20-channel SiRF-III-based GPS module at a sampling rate of 1 Hz. Gas concentration levels

are measured by three analog sensors, whose readings, along with the temperature, relative humidity, a time stamp, and GPS data, are stored in a 16 Mbit serial flash device, capable of holding up to 6 hours and 50 minutes of data without offloading the information. A 2x16 character LCD panel provides immediate visual feedback about the status of the system, including connected interfaces, GPS lock status, motion detection, sensor readings, and battery life, and an Intel 8051-based microcontroller controls all aspects of the system from battery charging to analog/digital conversions and the USB protocol. The node hardware architecture can be viewed graphically:

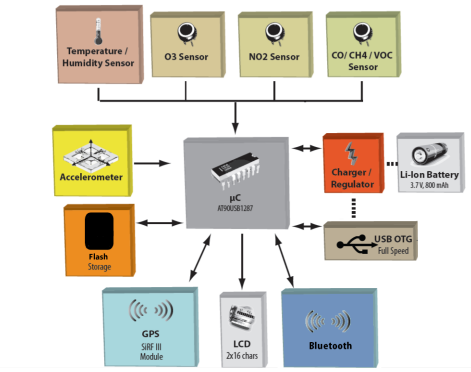


Figure 2: Graphical Hardware Architecture

Sensor Node Improvements

The sensor nodes described above worked extremely well for first generation prototypes; however, they did suffer from some limitations which needed to be accounted for before wide-scale deployment is possible. The following section outlines the changes and improvements that were made to the nodes to remedy these shortcomings.

The first problem was discovered while attempting to acquire a GPS lock with the digital LCD enabled. With the LCD detached from the circuitry, the GPS unit had very little trouble acquiring a lock; however, once the LCD was enabled, the GPS unit would often lose its lock, and if a lock had not already been acquired, it would often fail to ever lock onto a sufficient number of satellites to operate correctly. This seemed to be a problem with the LCD either producing too much electromagnetic interference between the GPS unit and

a satellite, or a problem with signal noise on the power supply grid increasing when the LCD was turned on. An oscilloscope verified that the board's +3.3V power grid sagged and became substantially more noisy when the LCD circuitry (including a charge pump to increase the LCD supply voltage to +5V) was connected. Since the ENABLE line of the GPS unit as well as its antenna power relies on a dependable +3.3V source, it was thought that the noisy signal could be disrupting the unit's ability to function correctly.

To fix the noisy power supply problem, and additional bypass capacitor was introduced to the system at the power supply's point of origin. The system already used a 10μF bypass capacitor at the source, but the positive terminal of an additional 2.2μF bypass capacitor (used at the charge pump input servicing the LCD power supply) was shorted via a wire to this terminal, providing parallel bypass capacitance of 12.2μF. This greatly decreased the power supply noise and enabled the GPS to retain its satellite lock once the LCD was connected to the circuitry, but only if it had already acquired a lock prior to enabling the LCD. If the LCD was turned on prior to the GPS acquiring a lock, there was still a high likelihood that it never would. Thus, it was decided that the electromagnetic field being produced by the LCD must be interfering with the GPS unit on the node. To overcome this, we simply purchased external GPS antennae which could be positioned far enough away from the LCD to avoid interference. This solved the locking problem in the prototypes, but in future versions of sensor nodes, GPS solutions will be required which provide better shielding from radio interference.

Firmware

All of the system components on each node are completely controlled by its Intel 8051-based microcontroller running firmware written in C using the Small Device C Compiler (SDCC) tool chain for compilation [7]. The benefit of using this method of compilation as opposed to a more commercial alternative (such as Keil) is that SDCC is open-source software, meaning that it is not only cheaper to produce firmware for these devices and keep them up to date, but also the source code can be released under the GNU Public

License [4], allowing researchers and developers to use the code in their own systems and even help to improve upon this existing code for the betterment of the entire system.

The main job of the code running on the microcontroller is to synchronously coordinate all of the responsibilities of the sensor node, including polling the gas sensors and communicating with base stations over Bluetooth or USB. The entire node runs at a software-defined frequency of 0.833 Hz, equivalent to 1 clock cycle every 1.2 seconds. At every cycle, the node reads the ambient temperature and relative humidity, polls the gas sensors, updates the status of the battery, USB, and GPS lock, stores this information into flash memory, and sends the data over Bluetooth or USB if applicable at the time. A software-defined watchdog timer is also implemented to ensure that the system does not lock up. The watchdog timer is reset in software at every clock cycle. Thus, if the system has frozen for any reason, the timer will not reset and will run out after approximately 10 seconds. At this time, an interrupt will fire, completely resetting the node (but not overwriting its flash memory contents), such that normal operation may proceed.

Another important aspect of the firmware is its ability to recognize when the unit is no longer in motion and turn off power-hungry components. A counter is initially set to 60 and decremented every clock cycle that the unit is stationary (as indicated by a logic '0' from the MEMS accelerometer). When the counter reaches 0 (after 72 seconds), the unit will power down the LCD, GPS, and Bluetooth devices to conserve power. Every time the unit is detected as being in motion, the counter is reset to 60 and all components are once again turned on.

The most important aspect of the firmware is, of course, its function in assuring that pollution information gets stored correctly in flash memory to be downloaded later as required. The data that is stored includes a maximum of 82 bytes of GPS data, all of the status information regarding the unit, as well as the actual atmospheric and pollution conditions, totaling 98 bytes of data. The flash memory unit is capable of storing 4,096 pages of data with each page containing 512 bytes [15]. Thus, each page can hold 5 data samples. This multiplied by 4,095 pages (the first page is reserved for memory content information) equals 20,475 samples of data that the flash memory can hold before overwriting previously logged data. That is equivalent to roughly 6 hours and 50 minutes before the node

must offload its information to a base station, either via Bluetooth or USB (both of whose host drivers are also implemented in the firmware).

CHAPTER III

SYSTEM ARCHITECTURE

Although the sensor nodes are the physical building blocks of the mobile air quality monitoring network, there are many other components which are necessary to implement this architecture on a global system scale. The most notable of these components is the server which is responsible for the processing, filtering, storing, and reconstructing of all the information, as well as the clients who are interested in accessing the data. Processes on the server combine data from individual sensors in similar physical regions into virtual sensors while adjusting for GPS inaccuracies. In addition to these components, the system also uses external sources of information to add to its accuracy and robustness, such as area maps, local weather and traffic information, and data from the 5,000 EPA sensors deployed across the United States [9].

From a central server point of view, data from sensors on the same node arrive at the server as part of a ”measurement tuple,” meaning a node samples all of its sensors simultaneously and generates a single tuple including these measurements, a time stamp, and the node's identifier. Data adapters then decompose the tuples into time-ordered streams for each sensed modality and can remove duplicates or flag suspicious readings. Currently, measurements are raw values and must be translated into scientific units (e.g. parts per million for CO concentrations). Each modality is calibrated using a different curve, and some modalities are calibrated first as they are required as input in the calibration of other modalities. Most notable are temperature and relative humidity, whose values greatly affect the measured levels of all gas concentrations using the analog sensors [26]. Some calibration curves are provided from the sensors' manufacturers while others will be periodically generated through measurements under controlled conditions in a laboratory.

Next, raw GPS readings are abstracted to a series of $<$location, duration$>$ pairs, using an innovative new approach described in Chapter V. After this, the node locations are embedded on a Google map, and data from all of the sensor platforms are combined

into a virtual sensor whose spatial and temporal sampling rate is equal to the sum of the rates of its constituents. This process is currently used to calibrate the sensors with one another, but will be used in the future to detect malfunctioning sensors, which can then be recalibrated using nearby mobile data as well as publicly available static data. The measurements from all sensors are used to indicate when and where gas concentrations fluctuate and to what extent these fluctuations occur. Finally, the discrete measurements from all sensors are used to reconstruct the overall pollution function for a given location using the process described in the following chapter.

Communications Methods

Since this system constitutes a large-scale wireless sensor network, operates autonomously, and is deeply embedded in the physical environment, the methods by which its constituent parts communicate with one another is of the utmost importance to the success of the network's implementation. This section will discuss such communications methods as necessary for smooth operation of the mobile pollution monitoring network.

Sensor Node Communications

It should be noted that future versions of the sensor node prototypes will replace the Bluetooth unit with a GSM modem, enabling units to transmit data over the mobile telephone networks. This will remove the need for any sort of base or docking stations and will decrease the complexity of the hardware as well as the power consumption of the device. Keep in mind that the research covered in the rest of this document assumes the presence of such a modem.

Raw data collected by the system's mobile sensing platforms are transmitted to the server network through one of the four following mechanisms:

1. **Periodic Upload**: Each of the mobile platforms periodically uploads the data it has collected since the last upload event. This upload period is determined by the platform's capabilities (i.e. energy and memory resources) and can be dynamically adjusted based on application requirements.

2. **Triggered Push**: The system can install trigger conditions to each of the mobile platforms whereby certain conditions (i.e. measured ozone concentrations exceed a certain threshold) will trigger the platform to perform an immediate upload. This will enable real-time alerts of potentially dangerous or hazardous situations.

3. **Triggered Pull**: The system also has the ability to dynamically pull data from specific mobile platforms. This may be necessary for example upon some external stimulus (e.g. a severe weather alert). This will also be used extensively for the web-based visualization of the data. For example, if a user wants to see the raw data coming from a specific node in real-time, triggered pulls will be used to ensure that the most recent and up-to-date information is available.

4. **Periodic Status Update**: Sensing platforms periodically report their status, including location, amount of data collected, and sensor status. This information is used to identify the sensors whose data must be pulled during an extreme event. Data from external sources (e.g. EPA sensors or traffic conditions) is also collected periodically or in response to extreme events.

Server Communications

The server in this system actually constitutes a server cloud, whereby the server processes as described in this paper do not actually run on one and only one centralized server, but rather on an interconnected distributed serving system. This increases the system's robustness and fault-tolerance while also decreasing the large bottleneck between potentially thousands to millions of clients trying to access system resources through only one server or server bank. The server system is distributed in the sense that each server array is connected, either directly or indirectly, to every other server array, such that data can be passed between servers in a manner similar to Internet routing [35].

The server system processes batches of incoming raw measurements as atomic tasks through a series of phases. The system stores the inputs and outputs of each stage so that failed steps can be rolled back if necessary. Only data tasks that successfully complete all phases are stored in each server's main database. Performing validation

and pre-processing avoids the cumbersome and resource intensive process of removing faulty data from the database. More abstract processing stages, such as context binding, interpolation, and so on, happen only after all the relevant data are available. Waiting for complete data sets before processing reduces the cost of sophisticated stages such as interpolation and improves the overall quality of the generated data.

All processing stages are implemented as user-defined functions and stored procedures, developed in the C programming language. The overall data management system operates using Microsoft SQL Server 2008, including its OpenGIS implementation. SQL Server 2008 is a good choice of data management software because it is a production-level Database Management System that offers high performance through clustering (the distribution of database information over several servers for redundancy and increased efficiency) [21]. Likewise, OpenGIS is an important database format component, since it allows other developers and programmers to access the stored information in a widely-available, open-source format [28].

Finally, the clustering of data between the servers will occur naturally on the basis of geographic location. There should be several servers located throughout the country which primarily service the sensing nodes and clients in their own sphere of influence. For example, all of the sensing units in Chicago will use their GSM modems to transmit data to the closest server bank to Chicago. This is extremely easy to do since the on-board GPS units of each device will be able to pinpoint their positions and choose the address of the correct server bank from an internal database. Likewise, web clients in the Chicago area will be routed to the closest server bank when they access the project web site. Finding a user's location based on IP address is an extremely well-defined process with many practical implementations already available, such as MaxMind's GeoLite City [27]. Thus, the traffic to and from each of the servers will be balanced according to their geographic location, lowering access and processing latencies. If, however, a user in Chicago wants to view the sensing data from another geographic location (New York, for example), the local server is acting as a cluster of the distributed whole; thus, it can fetch the required information from the New York server bank (either over the Internet or via dedicated lines) and store it in its own local cache in case it is needed again or until it expires.

Client Communications

The transfer of data between server and clients uses a new open-source platform called APE, short for ”AJAX Push Engine.” This engine makes use of an epoll-driven HTTP server written in C to allow data exchange between over 100,000 users per server via a web browser, without needing to reload and without relying on any external plug-ins other than Javascript, which comes standard in almost all graphical web browsers [2]. The great advantage of using APE for the purposes of this system is that it uses push technology instead of pull to deliver updates to subscribed clients as they appear on the server in real-time. This not only reduces the massive amount of network traffic required simply to request information every few seconds, but it also ensures that data is only sent when something interesting happens (i.e. when the data has changed or been updated, when new nodes have appeared, when an extreme event has occurred, etc.).

The client side of APE uses a Javascript framework to hook into its communications capabilities. In addition to any built-in Javascript methods, the APE Framework allows for new plug-in modules to be added to extend capability should any new unforeseen needs arise. In the context of the mobile pollution monitoring network, a single persistent connection is made over APE to the geographically-closest server whenever a user logs onto the project web site. At that time, the user is presented with a Google map which will be used to display sensing nodes and pollution data. The GPS coordinate bounds of the map’s viewport are communicated to the APE server, where they are used to subscribe the user to any nodes or pollution information falling within those bounds. Once this connection has been established, the only data that flows between server and client occurs when a node’s data has been updated or when the user changes the viewport and must subscribe to a new set of locations.

Since the APE framework has the ability to use a subscription method for client communications, an update for a single node could result in that update being propagated from a single server to numerous clients, similar to a data broadcast. The figure on the following page shows this:

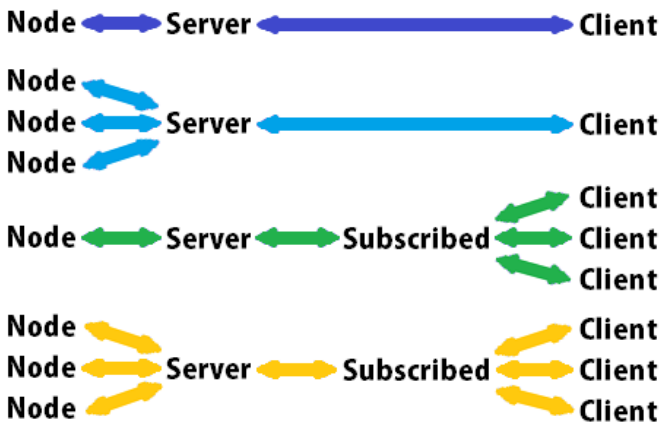


Figure 3: APE Information Dissemination

Also, since SQL Server 2008 is being used as the database system to drive APE, users have the ability to request (query) subsets of the total information available to them. In this way, a user simply viewing an overall sensor map is able to subscribe exclusively to sensor node location information without any of the accompanying gas sensor data. This greatly decreases the amount of traffic being sent over the network. If the user wants to access the raw data coming from one of the sensors on the map, then our visualization tools allow the user to select that sensor, thereby subscribing to its total data package. In this way, the only data that ever traverses the network is the data that the user has an interest in viewing, making the client-server communication lines as efficient and resource-friendly as possible.

Dynamic Calibration

A very large area of ongoing research for this network is a viable method of performing automatic dynamic calibration of the sensor nodes in realtime, such that the nodes do not have to be brought in to a servicing center to be calibrated on a fixed schedule or upon detection of any measurement anomalies. Since accurate gas concentration measurements require frequent recalibration and intensive accuracy checks, it is improbable that the network will initially be able to provide concentration data in terms of an absolute scale

(such as ppm). Instead, it will present data as concentrations relative to other concentration levels in the area. This, in itself, is a very valuable tool, enabling users to determine what areas in their vicinity contain the highest pollutant levels and what activities they may be doing to increase air pollution, such as idling their vehicle while waiting to pick up their children from school. This data also enables applications such as a ”Green Trip Planner” to be implemented, whereby a user can choose a route from point A to point B based on the least exposure to toxic pollutants. Likewise, once enough of these nodes are present on the network, we will be able to compare the data from the mobile sensing devices to calibrated data from static sensors in the near vicinity, and at that point, provide a better estimation of the absolute pollutant concentrations in a given area.

The important thing to note about such applications is that they rely on the sensor data from all of the mobile nodes to be calibrated to each other; that is, to the same external scale. This ensures that similar gas concentrations will result in multiple sensors producing the same measured levels. To ensure that this is the case, all sensors will be calibrated in a laboratory setting before being deployed to ensure that they initially match. Upon deployment, the gas measurements from each sensor will be periodically compared (in the server) to the average measurements of the rest of the sensors in their immediate vicinity. Once a sensor is found to be reporting values outside of some threshold percentage of the measured values of the rest of the sensors, it will be dynamically recalibrated such that its output matches that of the rest of the sensors in its region. This ensures that each of the sensors remain calibrated to one another and to the same scale.

Even though the gas sensors may remain calibrated to one another, this method could result in the entire system slowing dropping in sensitivity over time, such that the relative reported gas levels do not mean what they were originally intended to mean. To overcome this, sensors in the immediate vicinity of calibrated static sensors will automatically recalibrate themselves to the same external scale to which they were initially calibrated using the correct and accurate measurements from the static sensors. It has also been observed in laboratory experiments that the electrochemical gas sensors only become less sensitive and less responsive over time, never more sensitive. Thus, we can always assume that the most responsive sensor in a group (within reason, barring erroneous outliers from

malfunctioning sensors) will be the most accurately calibrated, and calibrate the remaining nearby sensors accordingly. This will ensure that the sensors remain calibrated both to one another as well as to a common external scale. The following flowchart depicts this process more clearly:

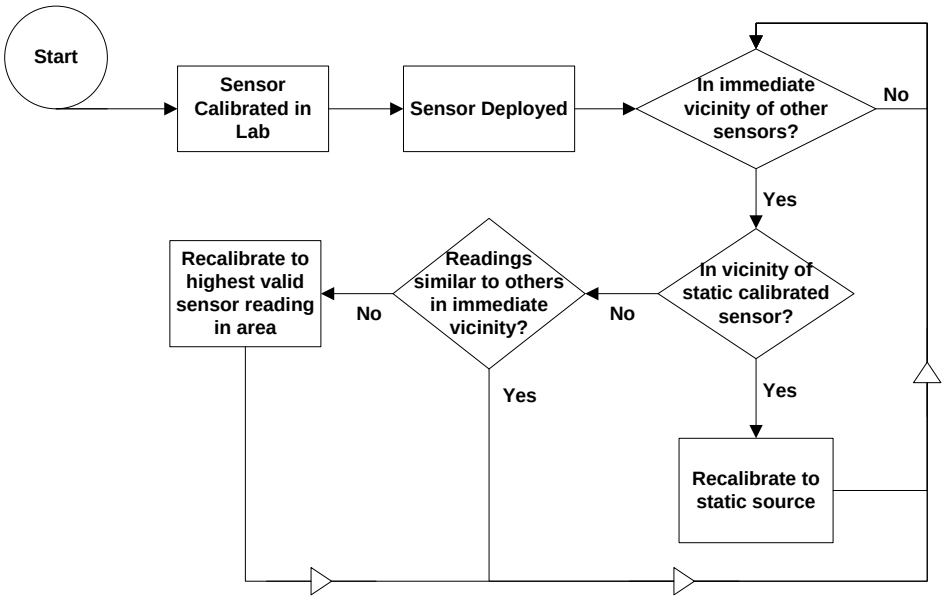


Figure 4: Dynamic Calibration Algorithm for Individual Sensors

CHAPTER IV

MODEL RECONSTRUCTION

The heart of this system comprises the need to reconstruct, process, and analyze pollution concentrations from data samples that are acquired with respect to both time and space. Specifically, our goal is to reconstruct a space-time signal $f \in L^2(\mathbb{R}^3 \text{x} \mathbb{R})$ from our knowledge of the samples $\{f(x_j, t_j) \colon x_j \in \mathbb{R}^3, t_j \in \mathbb{R}, j \in J\}$, where J is a countable index set. This is still an area of the project under intensive research; however, we have discussed several possibilities with mathematician Akram Aldroubi, and plan to find a nonlinear signal model that consists of a union of subspaces V_i instead of a single subspace V [11]. One of the advantages of finding a good nonlinear signal model is that the reconstructions will be sparse in terms of representations within the model [12, 25, 34]. This means that the reconstructed functions will be linear combinations of only a few basis or frame functions. Thus, storing or transmitting the reconstructed functions will be cheap and efficient. Moreover, the model search will implicitly allow us to take into account complex factors such as terrain and the fact that we are sampling almost exclusively near the main pollutant source, namely the traffic.

Given a signal model $\mathcal{M}$, our goal is to develop explicit reconstruction schemes. Obviously, real pollution concentration functions do not belong to the model $\mathcal{M}$ which is an approximation. Moreover, the samples are often corrupted by noise; thus, our algorithms must be robust to measurement noise and stable in terms of perturbations to $\mathcal{M}$. In addition, we require our algorithms to produce "good results," even if there are not enough samples to determine the function. Thus, the algorithms are required to be localized in space-time, producing good approximations where enough sample data are available [11]. Although this algorithm is still being developed, the rest of the research presented in the paper is not dependent on its completion, and so it will be viewed in black-box form throughout the rest of the work.

CHAPTER V

WEB-BASED VISUALIZATION

The most visible and integral part of this monitoring network for everyday users is the web interface used to access the vast quantity of data provided by the system in an intuitive and easy-to-use fashion. The main purpose of the web interface is to provide access to the reconstructed pollution signal. At the most basic level, the system is able to calculate a vector of contour points or a two dimensional array of data points for pollutants in an area at a specified resolution and time interval, calculate a vector of time instants when pollutants cross specified thresholds at some location, or return the time series of pollutants at a specified time resolution and location. On top of this basic set of queries, several web applications provide useful services to the end user, most notably web-based visualization.

GPS Smoothing

While GPS systems are extremely accurate when used in a situation where the GPS receiver is moving at a high velocity, this accuracy decreases substantially with a decrease in receiver speed, due to a lack of velocity information to aid in the modeling and estimation of GPS receiver position [14]. We have witnessed erroneous location measurements of up to 200 meters error when the receiver is stationary. Due to this inherent inaccuracy of GPS receivers, especially when the receiver is stationary, variable GPS coordinate readings will be generated with highly varying degrees of accuracy. Since a precise estimation of sensor location is essential in all areas of this system, it is important that these inconsistancies be handled gracefully, such that the most accurate location estimation possible may be obtained.

Cumulative Displacement Filter Overview

We have developed a new method of filtering and smoothing recorded GPS coordinates, especially in stationary and low-speed situations, based on a modified use of the

Cumulative Displacement Filter [16]. This filter relies on the assumption that the relative displacement between sets of coordinates are more accurate than the actual coordinates themselves in low-speed situations. Our modifications take advantage of the knowledge that GPS accuracy greatly increases with speed, and abrupt changes in location, azimuth, and speed are unlikely.

In our implementation, a variable-length history of GPS data is stored, such that the actual (albeit slightly time-delayed) position can be extrapolated from current, previous, and future measurements when the speed of the device decreases past a certain threshold or upon detection of obvious outliers. Essentially, the filter works by projecting positions onto a linear regression built from the normalized sum of displacements from each position in the history to the next. It should be noted that this filter cannot be used in real time, but instead requires a delay equal to the size of the filter history. This is acceptable in this application since data processing is done offline on a server, and the history only needs to be set to three data points to achieve maximum smoothing results. In addition to estimating positions at low speeds or with noisy signals, we have also added functionality to disallow changes in location when the device is thought to be stopped, judging by the fusion of inputs from the MEMS accelerometer, the speed estimation of the GPS receiver, and sporadic readings from the GPS unit. This greatly cleans up the signal at events such as traffic lights, stop signs, or weak GPS signals when the unit reports extremely inaccurate positions. Figure 5 shows the result (blue) of applying this filter to a noisy GPS signal (green):

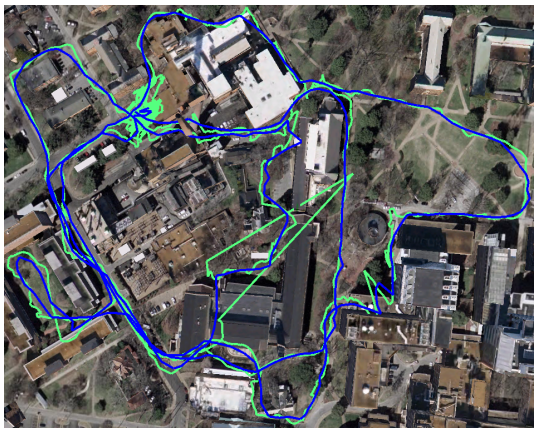


Figure 5: GPS Smoothing Results

Filter Implementation

The following flowchart describes the process by which our implementation of the GPS Smoothing Filter works:

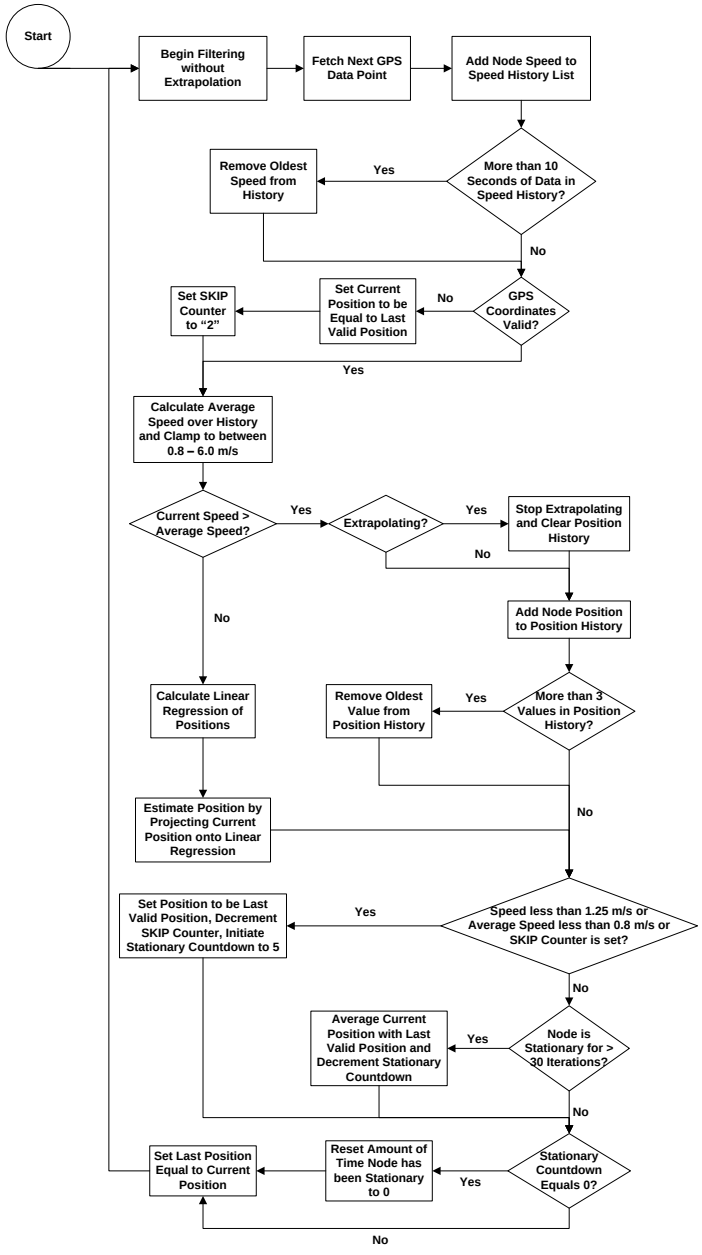


Figure 6: GPS Smoothing Filter Algorithm

Initially, a simple low-pass filter was tested to generate the aforementioned results. It was found, however, that a low-pass filter does not take into account typical GPS-receiver

behaviors, such as the possibility of losing GPS satellite locks, having highly erroneous measurements followed instantaneously by accurate measurements, or even stopping and then going in the reverse direction. As such, the smoothed results did not accurately reflect the ground truth, and oftentimes led to tracks that went off of roads and sometimes even into buildings. Our implementation has been found to give extremely accurate results with respect to an actual measured ground truth. In fact, the results as shown in Figure 5 so closely match the actual traversed ground truth for that track that a graphical representation primarily shows only a single track. The C++ code for this implementation can be found in Appendix A.

Flash-Based Web Client

Access to the pollution information in our databases occurs mainly via an innovative new flash-based web client, which enables users to not only access raw sensor data, but also visualize relative pollution functions in a manner similar to Weather Channel radar maps [8], interact with the map in an intuitive fashion to retrieve the desired data, use the data for several practical health-related applications such as a path planner to minimize exposure from travel source to destination, as well as gather targeted information about specific nodes or geographic areas over a variable time history. Our implementation of this client takes great pains to take into account the specific nature of the system to provide the most efficient, reliable, and robust interface possible. This section describes the various aspects of the web client, including its capabilities as well as the means by which it carries out its numerous tasks.

Map Functions

Google Maps Overview

The main and most important aspect of the web-based client is the Google Map used to display the sensor and pollutant information. The figure on the following page shows the web client as it appears when you first log onto the project web site:

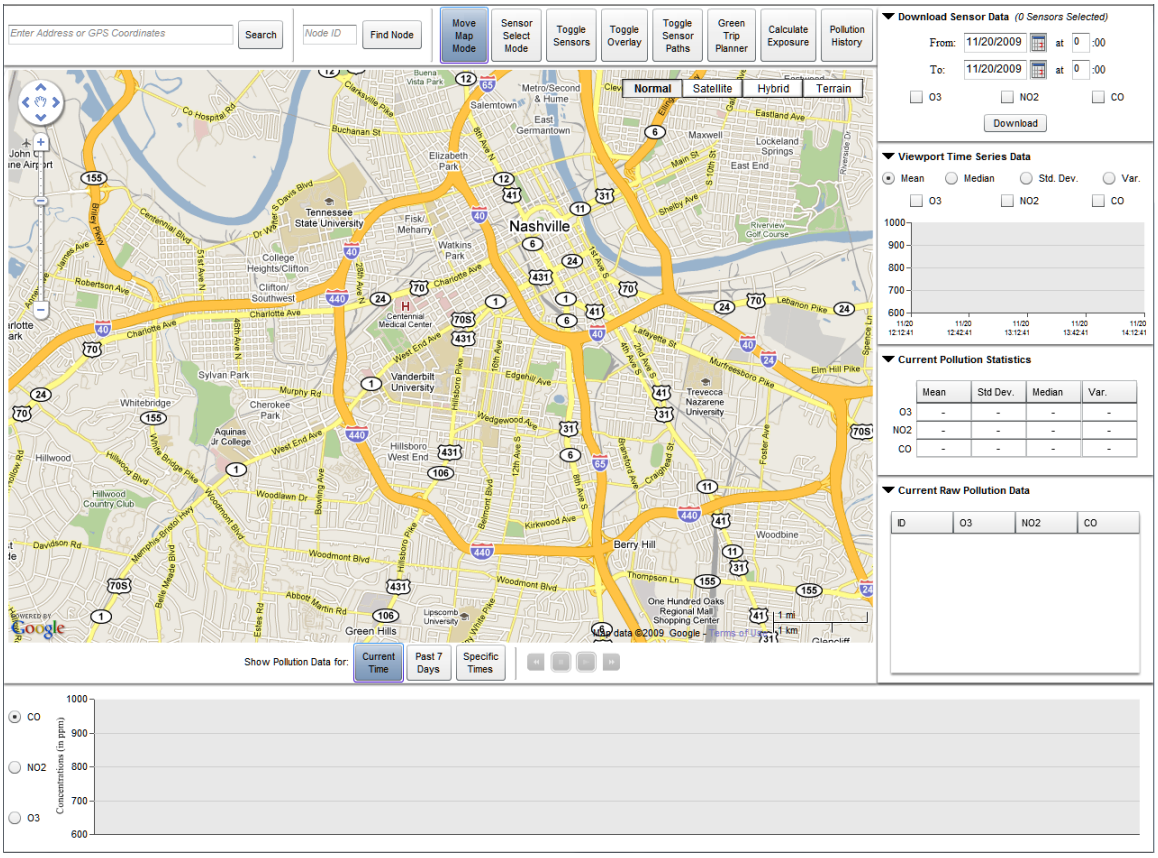


Figure 7: Flash-Based Web Client

All of the information that can be displayed by the client is highly customizable, down to the type and resolution of map used for displaying the sensors and pollution information. The user has the option to choose between four different map types [23], each having its own strengths when used for various applications:

1. **Normal**: This is the standard, most widely-used type of map. It is what a user typically sees when they request driving direction information from Google. Likewise, it is the most useful map type when viewing sensor positions, sensor path history, or when trying to map a route using the path of least pollution exposure.

2. **Satellite**: This type of map uses actual satellite imagery. It is useful when viewing pollution contour maps as it shows actual physical structures and can be used to determine what is causing variations in pollution concentrations in an area (for example, a paper mill producing elevated CO levels).

3. **Hybrid**: This adds street names and labels to the standard Satellite map and can be useful for finding your bearings when the Satellite map type is not descriptive enough.

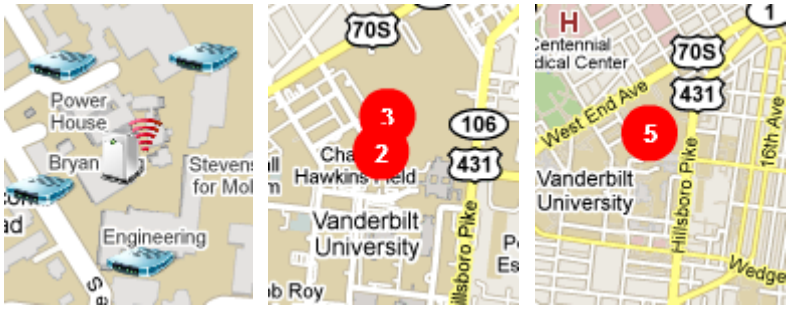
4. **Terrain**: This type of map uses a ”shades of grey”-based contour map to show the terrain structure of an area. It can be used in lieu of the Normal map, as its color sometimes makes it easier to visualize the pollution contour overlays than does the fairly colorful Normal map type.

In addition to these various maps, the user also has the ability to zoom in or out on the map to their desired viewing resolution [23].

Marker Clustering

Whenever there is one or more sensor marker located in the current viewing area, the marker shows up on the map as an icon. The nature of the icon varies depending on sensor type. Sensor types are predefined by the manufacturer or user and are transmitted to the server system during each of the ”Periodic Status Updates” as described in Chapter III. As such, each client receives information pertaining to the type of sensor (currently either ”static” or ”mobile”) being polled at the same time it receives the node ID number and location. If the sensor is one of our personally-deployed mobile sensors, it will have a different icon than any of the static sensors used by the EPA. This is useful for deciphering the accuracy of the data provided by the given node, and also provides room for further expansion, such that mobile nodes from several different companies or sources can all be deployed simultaneously and differentiated based on their icons. It should be noted, however, that the icons displayed for each sensor type are stored in the web client. Thus, any additional sensor types will require an upgrade to the code. Since the client is web-based, this only necessitates updating the flash file hosted on the servers with no additional maintenance steps required by the end-users. Figure 8 on the next page shows the two different icons currently in use as well as several ”clustered” markers which will be described next.

Once this system is deployed, it is likely that there will be many sensors located in close proximity to one another. Showing so many sensor icons at once will cause delays in the visualization system and will look cluttered, making it difficult to see what is happening at



(a) (b) (c)

Figure 8: Various Sensor Markers and Icons

a given location, let alone to select individual sensors out of a bunch. Likewise, when a user zooms out on a map, the scale of the map increases and the sensors move closer and closer together. To remedy this problem, we use a marker clustering approach as shown in parts (b) and (c) of Figure 8. This approach is embodied in a ”Marker Clusterer” component that analyzes the positions of all the sensor markers in the current viewport, and if the center of any markers are within 20 pixels of the center of any number of other markers, they are combined into one ”cluster marker” with the number of sensors contained in the cluster being displayed. To see the exact nodes that are contained in the cluster, the user can either move the cursor over the node where a text box will show the cluster contents, or zoom in such that the markers are sufficiently far apart to be displayed individually. In fact, all three portions of the above figure show the same five markers using different zoom levels.

All sensor markers are completely managed by the Marker Clusterer in this system, including the markers for single sensors. If the clustering algorithm deems the marker to be sufficiently far away from other markers, it takes care of adding the individual icon to the map. In addition to adding to the user-friendliness and visual aesthetics of the client, the marker clustering implementation also provides a better, more efficient way to manage the potentially vast number of markers present on the entire map. Instead of simply adding a sensor marker everywhere a physical sensor is located, the Marker Clusterer keeps track of all of the markers available to be drawn, but only adds the ones to the map that would currently be visible given the geographic bounds of the viewing region. For example, any

sensors located 10 miles to the west of the left boundary of the viewport in Figure 8 are completely unknown to the Google Maps system. Only when the user moves the viewport to the left 10 miles will the Marker Clusterer add them to the map (and remove the ones that are currently being shown). This keeps the Google Maps system from being bogged down by too much data, and also allows for 1000s of markers within the viewing area to be displayed with little to no noticeable delay to the user.

Sensor Node Selection

In addition to displaying the various locations of the sensor nodes, it is even more important to be able to select them such that their data can be retrieved and manipulated. To aid in this selection process, we have introduced two map modes to the client. The first mode is the standard Google Maps mode, simply called the ”Move Map Mode.” In this mode, the user has the ability to click anywhere on the map and drag it to change the bounds of the viewing area. This is the standard behavior defined by the Google Maps API [23]. Additionally, we have added the ability for users to click on individual sensors (or sensor clusters) in this mode to select them. Once selected, the node will remain selected until it is explicitly clicked again. Thus, the user can select multiple nodes, one at a time, and move the map around without disturbing the selections. A selected node is displayed with a blue oval around it.

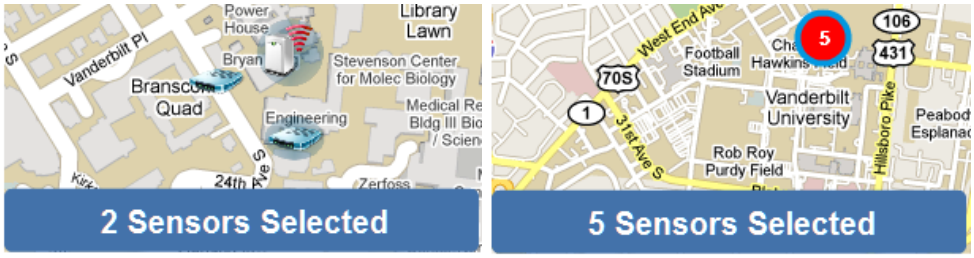


Figure 9: Single Sensor Node Selections

Selecting a cluster marker effectively selects all of the individual nodes within the cluster. This is achieved via the Marker Clusterer described in the previous subsection. Each

clustered marker acts as a container holding the individual markers within it; thus, any actions that occur on the cluster marker are simply forwarded to each of the single markers it contains. If a user selects a cluster node and then zooms in on the map until the individual nodes become visible, all of the nodes that belonged to the initial cluster will be selected. Likewise, if you select a bunch of individual nodes and then zoom out until these nodes form a cluster, the cluster will be displayed as selected. Whenever one or more nodes are selected, a blue box appears at the bottom left corner of the map informing the user how many sensors are currently selected.

The user may want to use this mode to view the details and data from individual sensors, but selecting individual nodes can become tedious, and there are times when it would be beneficial to select a range of sensors with one mouse motion. To facilitate this, we introduce another mode called ”Sensor Select Mode.” In this mode, the user can no longer move the map. Clicking on any point on the map while dragging the mouse creates a blue box. Any sensors lying within this blue box when the mouse button is depressed will be selected. Deselection works exactly the same way.

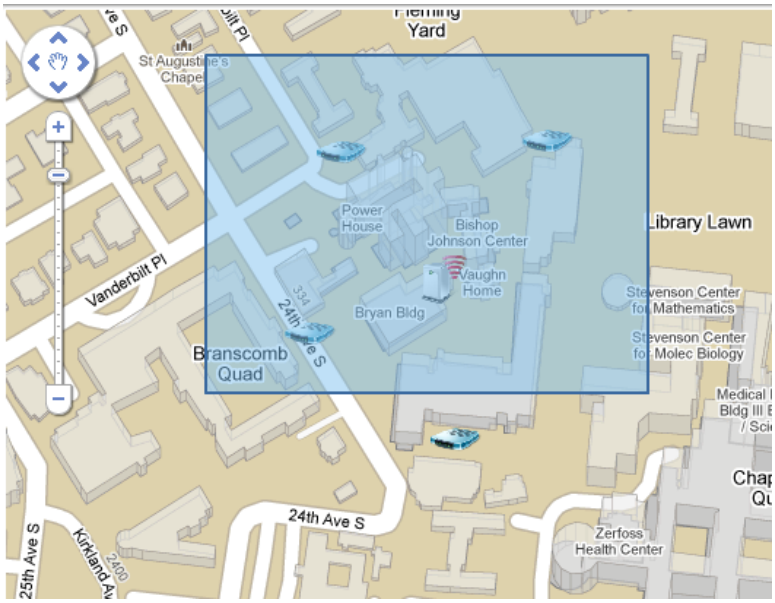


Figure 10: Multiple Sensor Node Selection

The underlying implementation of this selection mode again makes use of the Marker Clusterer component in a three step process. First, whenever a box is drawn on the map's

viewport, the pixel bounds of the selection are converted to a set of <latitude, longitude> coordinates using the Google Maps API [23]. The Marker Clusterer is then polled, requesting an array containing the locations of all the nodes currently visible within the viewport. Finally, each of these node locations is checked to see if it lies within the selection bounds as defined by the user. If so, a selection event is processed for that node exactly as if the user had simply clicked on the node's icon. This process repeats itself for all nodes in the viewport, mimicking the results if each of the selected nodes had been clicked individually. As in the "Move Map Mode," sensors can also be selected with simple mouse clicks. If the user wishes to move the map again, simply clicking the "Move Map Mode" button returns all normal functionality without disturbing the state of the selected markers.

Sensor Data

Data from the entire viewing area and each of the selected sensors is visible in the sidebar on the right side of the client. This is the most important part of the client for viewing and manipulating data and statistics. Please turn to the next page to see a figure depicting the sidebar area of the web client.

Each of the sub-panels in the sidebar is collapsible and expandable such that the user only has to view the data that is interesting or relevant. The top-most panel is used to download raw sensor data from the selected sensors. All that is required to use this function is a selection of some number of sensors on the map, the sensing modalities for which the user would like data, and the starting and ending times and dates of data to retrieve. A database request query is sent to a server over the persistent APE connection, asking for data from the selected nodes (using their node ID numbers) with additional filter parameters being the start and end dates specified. The data is returned in the form of a SQL query response. A dialog box will appear requesting a path to store information on the computer locally, and the returned sensor data will be written by the client to separate XML files for each selected node, with the XML fields of each file being:

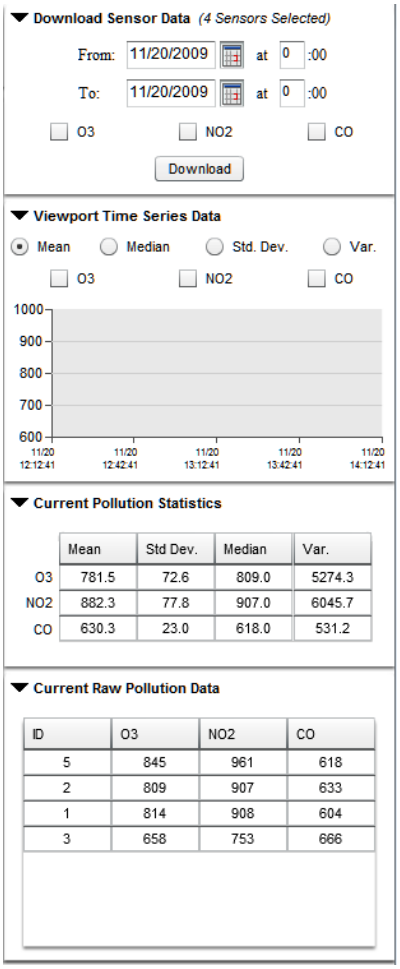


Figure 11: Web Client Sidebar

```
<data>
<datum date="05/19/2009" time="14:59:37" lat="36.145760"
       lng="-86.802587" temp="17.33" humidity="56.73" O3="698"
       NO2="396" CO="618" speed="0.40" azimuth="21.96" />
<datum>Etcetera for all data in specified time range />
</data>
```

This listing shows data for all three modalities, but the actual downloaded file will only include the requested data.

The next sub-panel is dependent upon the current viewing area, not the selected sensors. It displays the statistical time series data over the past two hours for the entire viewing area. The user selects which statistic and which modalities they are interested in, and a SQL query is sent over the APE connection, requesting sensor information over the past two hours, filtered by the geographic bounds of the current viewport. The requested

statistic is calculated from the returned data, and the graph displays the information, updating itself automatically whenever the viewport changes. So that the data is kept fresh, but unnecessary network traffic is not generated, the graph keeps a cache of its currently displayed data and updates every 30 minutes.

The third sub-panel shows the current statistical information for all sensing modalities for the range of selected sensors. This includes the mean, median, variance, and standard deviation for each pollutant. This is an extremely informational yet resource-economical function since it only requires the most recent sensor data from each of the selected nodes, and this can be obtained via a series of triggered pulls.

Finally, the last sub-panel shows the current raw data being reported by each of the selected sensors. Whenever a user selects a sensor, it is added to this list, and the raw O_3, NO_2, and CO levels are displayed in the table. Likewise, when a user deselects a sensor, it is removed from this list. It should be noted that all of these sidebar sub-panels are updated dynamically in realtime as new data arrives. Thus, the user can be assured that any of the data or statistics being shown represent the most current data available to the system.

Sensor Node Visibility and History

The actual map area is able to show a variety of overlays depending on the interests of the user. By default, all sensor node locations in the current viewport are displayed upon loading of the web client; however, other overlays require manual instantiation. All available overlays are able to be toggled on or off via their respective buttons on the top bar of the client. Currently, two other overlays are also available: a contour pollution overlay and a node path history overlay.

The contour pollution overlay is a visualization of the pollution function over a given area which, when fully implemented, will look very similar to a Weather Channel radar map [8]. The figure on the following page shows an example of this.

The redder areas of the overlay depict areas of higher pollution concentrations, while greener areas indicate less air pollution. When viewing this type of overlay, only one modality can be shown at a time. This is due to the colorful, contour nature of the overlay, making

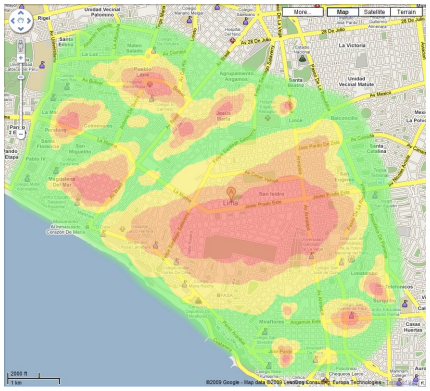


Figure 12: Sample Pollution Overlay

it impossible to differentiate between multiple overlays when shown simultaneously. These pollution maps will be generated by each individual web client using sparse data from the servers to feed our model reconstruction function, once it has been finalized.

The node path history overlay provides a way to visualize where a particular sensor has traveled over the past two hours. It can be used to show history information for all sensors in the viewport or only for selected sensors. In either case, the precise locations of the chosen sensors will be shown for the past two hours as different color line traces on the map. This history data is loaded via a simple SQL query to the web server for the locations of the indicated sensors over the past two hours. The resulting coordinates are fed directly into a ”Polyline” interface in the Google Maps API to produce the desired tracks. The following figure shows an example of such history tracks:

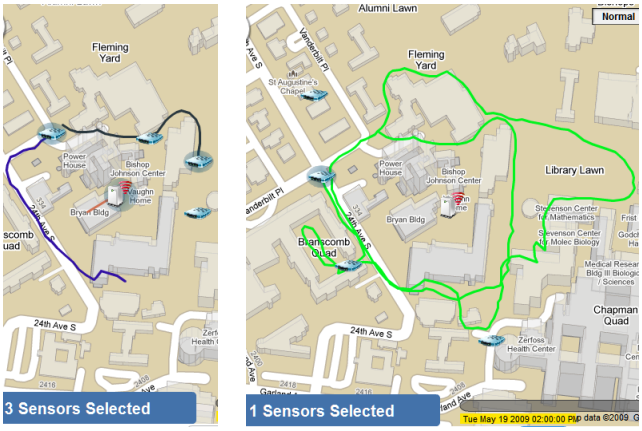


Figure 13: Node Path History Overlay

Realtime Raw Data Graph

Finally, the very bottom area of the web client displays a graph which is used to show the raw data for selected sensors in graphical form. The user selects a desired sensing modality, and the past two hours of history for each of the selected sensors will appear on the graph. As time progresses and newer information becomes available, the graph continually updates itself, scrolling to the left such that there are always two hours of data showing with the right-most data point being the most current for each of the selected sensors.

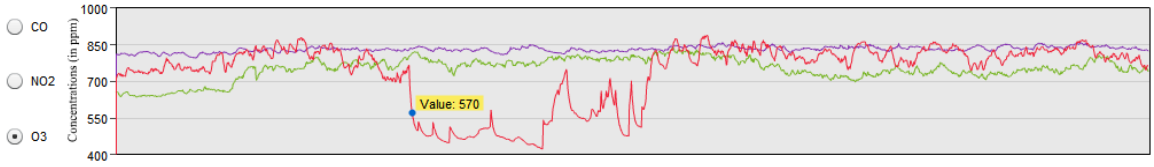


Figure 14: Web Client Graph Area

It is clear that each of the individual components of the web client reuses information from other areas, manipulating the data as necessary for the specified functionality. A great strength of this client is its ability to send requests for large amounts of data one time, then visualize the data in numerous ways from graphical to historical to statistical, while still remaining responsive and efficient. This is a primary goal of such a visualization framework, to present the available data in a variety of useful formats without overwhelming either the user, the system, or the resources required for such data-heavy processing and manipulation.

Data Retrieval

The greatest power and flexibility of this web client comes from the fact that all of the above functions can be used not only for current, realtime data, but also for a 7-day span of the most recent history or any user-defined span of times and dates in the past. As such, a user or analyst could look at and experience data from 2 years ago as if it were happening right now in real time.

The way this functionality works is similar to a video player. The user can select which

mode of operation they would like to use, indicating a desire to visualize data for the ”Current Time,” ”Past Seven Days,” or ”Specific Times.” Upon selection of ”Specific Times,” a dialog box appears prompting the user for the dates and times they would like to see:

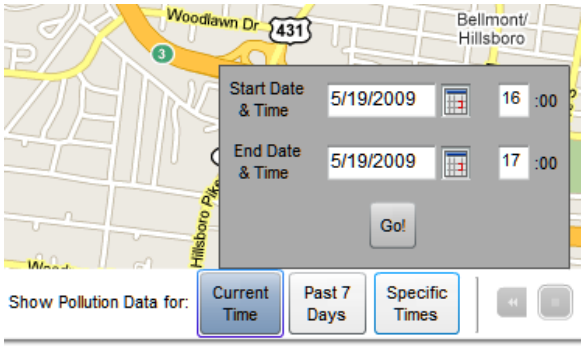


Figure 15: Choose History Dialog Box

The user is limited to viewing previous history in maximum chunks of seven days to prohibit overwhelming the system with too much data (since these types of operations are extremely resource-intensive). Whenever the user selects any mode other than ”Current Time,” a set of player controls appear at the bottom of the map, shown here:

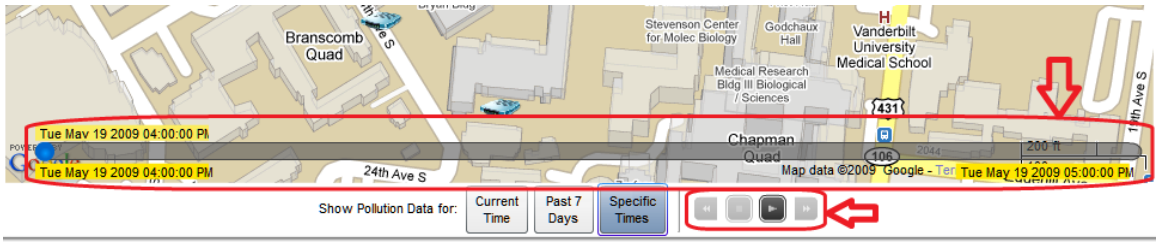


Figure 16: History Player Controls

These controls work just like any other standard video player controls. The labels at the bottom left and right of the timeline show the start and end times and dates, and the label above the tracking bead shows the current time and date being displayed by the web client. Pressing the play button causes the tracking bead to move to the right, playing the history in real time. If the user wants to see the history played in an accelerated fashion, they can press the ”faster” button to increase playback speed an unlimited number of times. Conversely, if they want to slow the play speed back to normal, they can press the ”slower” button as many times as they like until the play speed has returned to realtime. It should

be noted that playback cannot occur at speeds slower than the original realtime. If the user wants to directly skip ahead to some time, they can drag the bead forward or backward to reach the desired time and playback will resume from there. Finally, to stop playback, the ”stop” button can be pressed. Pressing play after stopping playback causes the player to continue showing the history from the time when the stop button was pressed.

The way that the player works is by making the web client think that the current time is something other than it is. If, for example, the user plays a self-defined history and drags the tracking bead to ”Nov. 3, 2009 at 11:28:38AM,” then the entire web-client system thinks it is operating at that specified date and time, and carries out its normal functionality under that assumption. As such, playback is just as efficient as realtime streaming of the data since it does not rely on any changes to the underlying code. The only thing that happens differently from realtime streaming is that all of the sensor locations for the nodes in the viewport over the entire history are loaded so that the user can skip around to different times on the timeline without having to wait for data to buffer. Actual pollution concentration data is still streamed in realtime like usual because the latency required for this is minimal, such that it can be done without creating lags or delays in system playback.

Whenever the viewing area is changed by the user, the history data reloads based on the new bounds and playback continues; thus, the amount of memory resources required is minimized by only storing the data currently visible in the viewport (exactly the same way that the Marker Clusterer only displays markers currently in the viewport). All client functionality is retained in any of these modes, including sensor node path histories and sidebar functions such as raw sensor data and current pollution statistics.

Additional Applications

In addition to the raw data, statistical, and visualization functionality of the web client described previously, the client also provides several innovative new applications for use by the general public and the health-conscious. The framework for these applications has been created, but their final implementation depends on the pollution model reconstruction

algorithm which has yet to be finalized. Thus, while the applications described in this section have been implemented, they assume the presence of such reconstruction techniques, and are not yet functional.

The first additional application available to users in this web-based client is a ”Green Trip Planner.” This application provides users with the ability to plan a driving route between two points based on the path of least exposure to pollutants. It has been designed to be extremely flexible, allowing users to compromise between pollution exposure and travel time for a happy medium. When the user clicks the ”Green Trip Planner” button, they are prompted to enter a starting address, ending address, and the pollutants they want to avoid. At that point, the planner will make use of the current pollution model to define areas to avoid between the two locations, leaving available only those areas with acceptable pollutant levels. It then uses the Google Maps' built-in ”driving directions” function to find a path from origin to destination though only those acceptable areas [24]. The directions are then displayed to the user along with a track on the map. If the user decides that the route is too long or inconvenient, they can click at various places on the map to add locations they explicitly want to drive through (for example, an interstate between two points). After selecting these points, the planner will then amend the directions to include the new points while retaining routes through as many of the pollution-safe zones as possible. The user can go through any number of iterations of this process until they are satisfied with the results, at which point, they will be given the option to print the directions or save them to a file.

The second additional application is a ”Past Exposure Estimation” application, which enables users to estimate past exposure to one or more pollutants given only a timed GPS track as an input. The GPS input comes in the form of a GPX (GPS eXchange) formatted file, which is a device-independent data format used by GPS devices to store information [5]. The estimator application will read in the contents of this file and map each of the GPS coordinates to a location in the pollution function for the given time. The output will be an XML file, whose contents are defined by the user and can contain pollution concentration information for each GPS coordinate, a list of times when the user experienced pollution

levels over a self-defined threshold level, or average concentration levels over specified time ranges.

Finally, our server databases have hooks programmed into them to allow other users and developers to access our data at a very low-level for use in their own applications and extensions.

CHAPTER VI

FUTURE WORK

As mentioned earlier, Bluetooth and USB are only useful communications methods if the sensors return to a base station. Mandating that every vehicle return to one or more stations where data can be downloaded is impractical. Moreover, we want the sensor data to stream in realtime, providing continuous up-to-date information to be used by our web client. As such, we will replace the Bluetooth unit with a GSM modem, enabling the units to transmit data over the mobile telephone networks [19]. It should be noted that this may increase system cost, since GSM solutions usually require purchasing a data plan with a cellular provider; however, relying on the availability of free WiFi access points is significantly less robust. Also, GSM coverage is excellent in most countries, and since there are already integrated GSM/GPS solutions, this modification will simplify the hardware [37, 29].

A key design decision for future prototypes will be the selection of gas sensors. Current gas sensors from e2v [3] have high sensitivity, but temperature dependance makes them difficult to use in this application. Also, they lose calibration very quickly when power-cycled over and over (as opposed to continuously running). Although we have devised ways to calibrate our prototypes based on experimental data and external pollution information, we are searching for sensors for the next prototype that are less temperature-dependent and whose outputs vary less from sensor to sensor, making dynamic calibration more manageable amongst a large number of nodes. Many other factors must also be considered, such as price, accuracy, and power consumption, and sensor lifespan should be maximized to cut down on future costs and alleviate the necessity of constant maintenance.

Fortunately, this is quite a dynamic area with many new products being produced, all of which provide variations on standard semiconductor gas sensing technology. In addition, we have found emerging technologies which show promising results in the realm of infrared gas sensors [32, 33], as well as a newly-released product from the VTT Technical Research

Centre of Finland [6] that makes use of a MEMS-based photo-acoustic gas sensor. We also plan to add CO_2 and SO_2 sensing capabilities to the system.

Additional health risks arise from the presence of particulate matter (PM) in the air which can accumulate in the respiratory system. Particles less than 10 micrometers in diameter (PM10) and fine particles less than 2.5 micrometers (PM2.5) are among the most harmful pollutants [30]. Unfortunately, even the cheapest PM sensors cost thousands of dollars. Instead of integrating one into our sensor nodes, we will provide an external interface for pluggable PM sensors. This way, some of the deployed nodes can use this capability, but we can keep the cost of the system down.

Currently, the main research challenge is designing an efficient implementation to reconstruct a pollution function from sparse and irregularly-sampled sensor measurements in a 4-dimensional database table. Such a reconstruction algorithm is essential to creating contour-like pollution maps as well for applications which need this sort of information to provide higher-level services (such as green trip planning and past exposure estimation).

CHAPTER VII

RELATED WORK

Pollution awareness has increased substantially over recent years, motivating several projects that use mobile platforms to collect air quality measurements [22, 31, 13, 17]. One of the primary differences between these earlier works and ours is that the earlier mobile sensors were not networked, but rather relied on the manual download of measurements [22, 31]. The PEIR project from UCLA uses cell phones to collect air quality data from users participating in the system. Our system places air pollution sensors on vehicles. This decision allows us to use not only more accurate sensors, but also sensors to measure a wider spectrum of air quality parameters such as particulate matter, without having to worry about power consumption or other resource constraints that make cell phone based approaches impractical. Additionally, our system is better-suited for pollution estimation since vehicles spend most of their time outdoors covering a wide geographic area when in use. Cell phones tend to stay in a person's pocket who is usually located indoors where pollution levels may deviate greatly from outdoor levels.

The Common Sense project from Intel Research works similarly to the system in our project [13]. It uses air pollution sensors mounted on vehicles, delivering measurements via GSM text messages. Likewise, a conglomeration of universities in the United Kingdom are working together on a platform that is most similar to ours, creating a pollution sensing network using a variety of mobile sensing platforms including ”smart-dust,” mobile motes, static sensors, and cell phones [18]. The system outlined in this document, however, takes the approach to a higher level. We are not only developing algorithms for the reconstruction of air pollution information from irregular and sparse measurements, but also devising methods of visualization such as contour maps as opposed to simple discrete observations. In addition, our system implements practical applications which provide typical system users with valuable and accessible data that promises to impact everyday life, not only the needs of air pollution specialists or experts. Furthermore, we will be using the

cellular network as a mode of data transmission decoupled from the use of an actual cell phone, since this method is almost always available and offers consistent service in both the United States and other foreign countries, as well as developing countries.

CHAPTER VIII

CONCLUSIONS

The overarching goal of this project is to dramatically increase the resolution of air pollution information and maximize its impact on public life. Our pollution monitoring system operates deeply embedded in the physical environment. Irregular spatiotemporal samples are used to reconstruct an overall pollution model affected by emission sources, weather, terrain, traffic patterns, and other factors. We have designed mobile nodes to sense three known air pollutants, O_3, NO_2, and CO, as well as ambient environmental conditions and communicate this data to a central server for the purpose of providing continuous realtime data feeds over a web interface.

The system provides an intuitive method of data retrieval using web-based visualization with a number of novel applications making use of the high-resolution data. Users have the ability to not only download raw sensor data from any number of mobile sensing devices, but also to stream pollution information in real time, visualize this data in easy-to-understand contour-like pollution maps, and gather statistical information about pollutants over a specified spatiotemporal region. In addition to being able to do this with current realtime data, users can also access historical pollution information using the same interface in a manner similar to a standard video player, allowing researchers to perform historical analyses using our data.

The highest level of access to our pollution information comprises two novel applications, one of which can be used to estimate an individual's past exposure to a pollutant using only a single timed GPS track, and another which provides a route planning service to minimize a person's exposure to a given set of pollutants. Although some aspects of this project are still in research and development, we have outlined a solid framework for implementing a mobile air pollution monitoring network with concrete applications in the real world. It is now up to the public to utilize the resources we have provided to not only benefit their own personal level of health, but also to increase overall awareness of the

societal impact of air pollution, so that together, we can work to promote a cleaner, safer environment.

CHAPTER A

GPS SMOOTHING FILTER IMPLEMENTATION

CDF.h:

```
1   // Based on "Cumulative Displacement Filter" by Julien Cayzac.
2   // Released under a Creative Commons 2.5 Attribution license.
3
4   #ifndef __CUMULATIVE_DISPLACEMENT_FILTER_H__
5   #define __CUMULATIVE_DISPLACEMENT_FILTER_H__
6
7   #include <cstring>
8   #include <list>
9
10  struct PositionStruct
11  {
12      // Constructors
13      PositionStruct() : longitude(0.0), latitude(0.0), azimuth(0.0),
14          speed(0.0), timestamp(0) {}
15      PositionStruct(const PositionStruct &o) : longitude(o.longitude),
16          latitude(o.latitude), azimuth(o.azimuth), speed(o.speed),
17          timestamp(o.timestamp)
18      {
19          memcpy(date, o.date, sizeof(date));
20          memcpy(time, o.time, sizeof(time));
21          memcpy(temperature, o.temperature, sizeof(temperature));
22          memcpy(humidity, o.humidity, sizeof(humidity));
23          memcpy(O3, o.O3, sizeof(O3));
24          memcpy(NO2, o.NO2, sizeof(NO2));
25          memcpy(CO, o.CO, sizeof(CO));
26      }
27
28      // Member variables
29      double longitude, latitude, azimuth, speed;
30      unsigned int timestamp;
31      char date[20], time[20], temperature[15], humidity[20];
32      char O3[12], NO2[12], CO[12];
33  };
34
35  class CumulativeDisplacementFilter
36  {
37      public:
38
39          // Constructor
40          CumulativeDisplacementFilter(unsigned int historySize);
41
42          void reset(void);
43          void filter(PositionStruct &pos);
```

```
44
45      private:
46
47          // Disallow use of this constructor
48          CumulativeDisplacementFilter(void){}
49
50          std::list<PositionStruct> m_Speeds;
51          std::list<PositionStruct> m_History;
52          unsigned int mui_historySize;
53          bool mb_extrapolating;
54          bool mb_lastPositionAvailable;
55          unsigned int mui_skipNext;
56          PositionStruct m_lastPosition;
57          bool mb_stationary;
58          int mi_stationaryCount;
59          int mi_stationaryCountdown;
60  };
61
62  #endif      // #define __CUMULATIVE_DISPLACEMENT_FILTER_H__
```

CDF.cpp:

```
 1  #include <cmath>
 2  #include "CDF.h"
 3
 4  #define PI 3.14159265358979323846
 5
 6  // Helper conversion functions
 7  static const double degreeArcLength = 111226.29991434248924368723;
 8  static const double degreeArcLengthRec = 0.00000899067936962857;
 9  void convertWGS84ToSI(double dLongitude, double dLatitude,
10                        double &dCoordX, double &dCoordY)
11  {
12      if (abs(dLatitude) > 85.0)
13          dLatitude = ((dLatitude < 0.0) ? -85.0 : 85.0);
14
15      dCoordY = dLatitude * degreeArcLength;
16      dCoordX = dLongitude * degreeArcLength *
17          cos(dLatitude * 0.01745329251994329576);
18  }
19  void convertSIToWGS84(double dCoordX, double dCoordY,
20                        double &dLongitude, double &dLatitude)
21  {
22      dLatitude = dCoordY * degreeArcLengthRec;
23      dLongitude = (dCoordX * degreeArcLengthRec) /
24          cos(dLatitude * 0.01745329251994329576);
25  }
26
27  // Constructor
28  CumulativeDisplacementFilter::
29      CumulativeDisplacementFilter(unsigned int historySize) :
30            mui_historySize(historySize), mb_extrapolating(true),
```

```
31              mb_lastPositionAvailable(false), mui_skipNext(0),
32              mb_stationary(false), mi_stationaryCount(0)
33      {}
34
35      // Resets filter history
36      void CumulativeDisplacementFilter::reset()
37      {
38          m_History.clear();
39      }
40
41      // Performs actual filtering
42      void CumulativeDisplacementFilter::filter(PositionStruct &pos)
43      {
44          // Remove extraneous speeds longer than 10 seconds ago
45          PositionStruct saved(pos);
46          m_Speeds.push_back(pos);
47          while (((long)pos.timestamp -
48              (long)(*m_Speeds.begin()).timestamp) > 10000)
49                  m_Speeds.pop_front();
50
51          // GPS Lock lost, assume last known position and skip the next
52          // 2 valid GPS positions to ensure accurate location estimation
53          if ((pos.latitude == 0.0) || (pos.longitude == 0.0))
54          {
55              pos.latitude = m_lastPosition.latitude;
56              pos.longitude = m_lastPosition.longitude;
57              pos.speed = m_lastPosition.speed;
58              pos.azimuth = m_lastPosition.azimuth;
59              mui_skipNext = 2;
60          }
61
62          // Calculate average speed
63          double avg_speed = 0.0f;
64          std::list<PositionStruct>::iterator dit = m_Speeds.begin();
65          while (dit != m_Speeds.end())
66              avg_speed += (*(dit++)).speed;
67          avg_speed /= (double)m_Speeds.size();
68
69          // Clamp speed between 0.8 and 6.0 m/s
70          if (avg_speed > 6.0)
71              avg_speed = 6.0;
72          else if (avg_speed < 0.8)
73              avg_speed = 0.8;
74
75          // Stop extrapolating if moving sufficiently fast
76          if (pos.speed >= avg_speed)
77          {
78              if (mb_extrapolating)
79                  m_History.clear();
80              mb_extrapolating = false;
81              m_History.push_back(pos);
82
83              if (m_History.size() > mui_historySize)
84                  m_History.pop_front();
```

```
85        }
86        else if (!m_History.empty())  // If history is not empty
87        {
88           // Get linear regression of positions
89           double sx = 0.0, mx = 0.0, sy = 0.0, my = 0.0;
90           std::list<PositionStruct>::iterator it = m_History.begin();
91           long deltaT = (long)((*it).timestamp);
92           while (it != m_History.end())
93           {
94              PositionStruct &posi = *it;
95              ++it;
96              deltaT = (long)pos.timestamp - deltaT;
97              double azimuth = 90.0 - posi.azimuth;
98              if (azimuth >= 360.0)
99                 azimuth -= 270.0;
100             if (azimuth > 180.0)
101                azimuth = 360.0 - azimuth;
102             azimuth *= 0.01745329251994329576;
103             double multiplier = posi.speed * (double)deltaT * 0.001;
104             sx += cos(azimuth) * multiplier;
105             sy += sin(azimuth) * multiplier;
106             double ax, ay;
107             convertWGS84ToSI(posi.longitude, posi.latitude, ax, ay);
108             mx += ax;
109             my += ay;
110          }
111
112          // Normalize supporting vector [sx, sy]
113          double ilen = sx*sx + sy*sy;
114          if (ilen >= 0.00001)
115          {
116             ilen = 1.0 / sqrt(ilen);
117             sx *= ilen;
118             sy *= ilen;
119
120             // [mx, my] are the position's sums. Convert to
121             // average position
122             double icount = 1.0 / (double)m_History.size();
123             mx *= icount;
124             my *= icount;
125
126             // Position in linear regression's local system
127             // ([mx, my], [sx, sy], [-sy, sx])
128             double x, y, posx, posy;
129             convertWGS84ToSI(pos.longitude, pos.latitude, posx, posy);
130             x = posx - mx;
131             y = posy - my;
132             double dot = x*sx + y*sy;
133             mb_extrapolating = true;
134
135             // Build new position
136             x = mx + (dot * sx);
137             y = my + (dot * sy);
138             double lon, lat;
```

```
139             convertSIToWGS84(x, y, lon, lat);
140             pos.longitude = lon;
141             pos.latitude = lat;
142
143             // Build new azimuth from linear regression supporting
144             // vector [sx, sy]
145             double azimuth = 90.0 - (((sy < 0.0) ? -acos(sx) :
146                acos(sx)) / 0.01745329251994329576);
147             if (azimuth < 0.0)
148                azimuth += 360.0;
149             pos.azimuth = azimuth;
150          }
151       }
152
153       // Low motion (GPS speed irrelevant)
154       if (mb_lastPositionAvailable)
155       {
156          double ax, ay, dx, dy;
157          convertWGS84ToSI(pos.longitude, pos.latitude, ax, ay);
158          dx = ax;
159          dy = ay;
160          convertWGS84ToSI(m_lastPosition.longitude,
161             m_lastPosition.latitude, ax, ay);
162          dx -= ax;
163          dy -= ay;
164          dx = dx*dx + dy*dy;
165
166          // If this position has barely moved from last valid position,
167          // or is this speed is less than 1.25 m/s, or if the average
168          // speed over the last 10 seconds is less than 0.8 m/s, or if
169          // we are still skipping valid points due to the loss of the
170          // GPS lock, set current position to be last known position
171          if ((dx < 200.0) || (pos.speed < 1.25) ||
172             (avg_speed <= 0.8) || (mui_skipNext != 0))
173          {
174             ++mi_stationaryCount;
175             mi_stationaryCountdown = 5;
176             unsigned int timestamp = pos.timestamp;
177             double speed = pos.speed;
178             char date[20], time[20], temperature[15], humidity[20];
179             char O3[12], NO2[12], CO[12];
180             memcpy(date, pos.date, sizeof(date));
181             memcpy(time, pos.time, sizeof(time));
182             memcpy(temperature, pos.temperature, sizeof(temperature));
183             memcpy(humidity, pos.humidity, sizeof(humidity));
184             memcpy(O3, pos.O3, sizeof(O3));
185             memcpy(NO2, pos.NO2, sizeof(NO2));
186             memcpy(CO, pos.CO, sizeof(CO));
187
188             pos = m_lastPosition;
189             pos.timestamp = timestamp;
190             pos.speed = speed;
191             memcpy(pos.date, date, sizeof(date));
192             memcpy(pos.time, time, sizeof(time));
```

```
193             memcpy(pos.temperature, temperature, sizeof(temperature));
194             memcpy(pos.humidity, humidity, sizeof(humidity));
195             memcpy(pos.O3, O3, sizeof(O3));
196             memcpy(pos.NO2, NO2, sizeof(NO2));
197             memcpy(pos.CO, CO, sizeof(CO));
198
199             if (mui_skipNext != 0)
200                --mui_skipNext;
201          }
202          // If node has been stationary for 30 or more iterations and is
203          // now moving, average previous position with new positions to
204          // account for missing coordinates if GPS lock was lost
205          else if (mi_stationaryCount >= 30)
206          {
207             pos.azimuth = (pos.azimuth + m_lastPosition.azimuth) / 2.0;
208             pos.latitude = (pos.latitude + m_lastPosition.latitude) /
209                2.0;
210             pos.longitude = (pos.longitude + m_lastPosition.longitude)
211                / 2.0;
212             pos.speed = (pos.speed + m_lastPosition.speed) / 2.0;
213             --mi_stationaryCountdown;
214          }
215
216          // Node has been in motion for last 5 iterations, resume
217          // normal operation
218          if (mi_stationaryCountdown == 0)
219             mi_stationaryCount = 0;
220       }
221
222       m_lastPosition = pos;
223       mb_lastPositionAvailable = true;
224    }
```

BIBLIOGRAPHY

[1] AIRNow: Quality of air means quality of life. Available at http://airnow.gov/, 2009.

[2] APE: AJAX Push Engine. Available at http://www.ape-project.org/, 2009.

[3] e2v Gas Sensors. Available at http://www.e2v.com/products-and-services/sensors/gas-sensors/, 2009.

[4] The GNU General Public License. Version 3, Available at http://www.gnu.org/licenses/gpl.html, 29 June 2007.

[5] GPX: the GPS Exchange Format. Available at http://www.topografix.com/GPX/1/1/, 2007.

[6] Miniaturised photoacoustic gas sensor. Available at http://fp7minigas.openinno.fi:8080/bin/view/Main/, 2009.

[7] SDCC: Small Device C Compiler. Available at http://sdcc.sourceforge.net/, 2009.

[8] The Weather Channel, Interactive Map. Available at http://www.weather.com/weather/map/interactive/, 1995-2009.

[9] Environmental Protection Agency. Air pollution data sources. Available at http://www.epa.gov/air/airpolldata.html, May 2009.

[10] Environmental Protection Agency. Air Quality Index (AQI) - A guide to air quality and your health. Available at http://www.airnow.gov/index.cfm?action=aqibasics.aqi, 2009.

[11] Akram Aldroubi. Personal Communication with Akram Aldroubi, 2009.

[12] Akram Aldroubi, Carlos Cabrelli, and Ursula Molter. Optimal non-linear models for sparsity and sampling. *Journal of Fourier Analysis and Applications*, 14:793–812, 2008.

[13] P. M. Aoki, R. J. Honicky, A. Mainwaring, C. Myers, E. Paulos, S. Subramanian, and A. Woodruff. Common sense: Mobile environmental sensing platforms to support community action and citizen science (demonstration). In *Adjunct Proceedings, Ubicomp 2008*, September 2008.

[14] Dale Arden. MEMS/GPS Kalman Filter. Defense Research and Development Canada. May 2007.

[15] Atmel. AT45DB161D 16-megabit DataFlash Technical Sheet, 2007.

[16] J. Cayzac. The cumulative displacement filter. Available at http://julien.cayzac.name/code/gps/, 2006.

[17] UCLA CENS. Personal environmental impact report. Available at http://peir.cens.ucla.edu/, 2009.

[18] Imperial College, University of Cambridge, University of Leeds, University of Southampton, and Newcastle University. Message project: Mobile environmental sensing system across grid environments. Available at http://bioinf.ncl.ac.uk/message/, 2008.

[19] Internation Engineering Consortium. Global System for Mobile Communication (GSM) White Paper. Available at http://www.iec.org/online/tutorials/gsm/index.asp, 2007.

[20] United Nations Population Division. World urbanization prospects: The 2007 revision population database, Sep 2009. Available at http://esa.un.org/unup/index.asp.

[21] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. A database interface for clustering in large spatial databases. In *Proceedings of 1st International Conference on Knowledge Discovery and Data Mining*, 1995.

[22] M. Ghanem, Y. Guo, J. Hassard, M. Osmond, and Richards. M. Sensor grids for air pollution monitoring. In *Proceedings of the 3rd UK e-Science All Hands Meeting*, 2004.

[23] Google. Google Maps API for Flash. Available at http://code.google.com/apis/maps/documentation/flash/, 2009.

[24] Google. Google Maps API for Flash: Directions. Available at http://code.google.com/apis/maps/documentation/flash/reference.html#Directions, 2009.

[25] R. Gribonval and M. Nielsen. Sparse representations in unions of bases. In *IEEE Transactions on Information Theory*, volume 49, pages 3320–3325, Dec 2003.

[26] Shigeki Hirobayashi, Haruhiko Kimura, and Takashi Oyabu. Dynamic model to estimate the dependence of gas sensor characteristics on temperature and humidity in environment. *Sensors and Actuators B: Chemical*, 60(1):78–82, Nov 1999.

[27] MaxMind. Geolite city. Available at http://www.maxmind.com/app/geolitecity, 2009.

[28] Inc. Open Geospatial Consortium. Opengis standards. Available at http://www.opengeospatial.org/standards, 2009.

[29] Advanced Wireless Planet. GSM/GPS Module. Available at http://www.gsm-modem.de/trizium.html, 2009.

[30] C.A. Pope, 3rd, M.J. Thun, M.M. Namboodiri, D.W. Dockery, J.S. Evans, F.E. Speizer, and C.W. Heath, Jr. Particulate air pollution as a predictor of mortality in a prospective study of U.S. adults. *American Journal of Respiratory and Critical Care Medicine*, 151(3):669–674, Mar 1995.

[31] Paul Rudman, Steve North, and Matthew Chalmers. In *Proceedings of the UK-UbiNet workshop on eScience and ubicomp*, 2005.

[32] SensorChip. CO2 Gas Sensor. Available at http://photonics.icxt.com/uploads/files/Datasheets/SensorChipCO2-ds.pdf, 2009.

[33] Siemens. Gas sensor research. Available at http://w1.siemens.com/innovation/en/publikationen/publications_pof/pof_fall_2004/sensors_articles/gas_sensors.htm, 2004.

[34] J.A Tropp. Greed is good: Algorithmic results for sparse approximation. In *IEEE Transactions on Information Theory*, volume 50, pages 2231–2242, Oct 2004.

[35] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Toward a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1), Jan.

[36] Peter Volgyesi, Andras Nadas, Xenofon D. Koutsoukos, and Akos Ledeczi. Air quality monitoring with sensormap. *IPSN*, pages 529–530, 2008.

[37] GSM World. GSM Coverage Maps. Available at http://www.gsmworld.com/roaming/gsminfo/index.shtml, 2009.