Design Considerations in Developing a Mobile Application for Scalable and Decentralized Publish/Subscribe-based Weather Alert System

Violetta Vylegzhanina David Harmon Brett Aniruddha Gokhale
Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA
{violetta.vylegzhanina,david.h.brett,a.gokhale}@vanderbilt.edu

Abstract

This paper describes our experience developing a mobile, cloud-based weather alert system, which was motivated by the need to overcome the limitations in an existing, centralized severe weather warning system at Vanderbilt University. We faced different design choices, integration challenges and a substantial learning curve in realizing a decentralized, scalable, and easy-to-use solution. In our solution, Android devices in the vicinity of the hazardous weather event can post timely data to a server, including the approximate location of the severe weather incident, and where the server, in turn, can distribute messages scalably to relevant Android devices that are in the potential path of the weather incident. Although, the use case for this research was severe weather alerting, the basic patterns and design choices used in the mobile application design can be used in many other related scenarios.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Architectures; C.2.4 [*Computer Communication Networks*]: Distributed Systems

General Terms Design, Management

Keywords Mobile app; real-time weather alert; Android; decentralized and scalable; design considerations.

1. Introduction

The work presented in this paper is motivated by the limitations we observed with an existing severe weather warning system at Vanderbilt University, which is a centralized solution that sends out weather notifications to all users who are registered with the system. There are major limitations with this solution. First, the centralized nature of the system adversely impacts scalability. Second, because alerts are sent to all registered users, a user who is currently not in the vicinity of hazardous weather may still receive an alert, thereby resulting in false alarms for that user.

The purpose of our research was thus to overcome the limitations in the existing weather warning system by investigating a solution that will be decentralized, scalable, and easy-to use as well as provide real-time notifications while minimizing false alarms.

Given the proliferation of mobile devices, it was appropriate for us to design a mobile app-based solution. Due to multiple integration challenges we faced and described in this paper, we did not consider a multi-platform solution yet. Instead, our solution is built on the Android platform. In our solution, Android devices that are in the proximity of the severe weather have the ability to post the approximate location of a severe weather to a server, which in turn distributes the messages scalably to selected devices that are in the path of the severe weather event, and eliminate false alarms.

This paper makes the following contributions.

- It focuses on describing our design, the choices we faced in the design decisions, the challenges we faced in integrating different technologies, and the learning curve we incurred in understanding the different technologies.
- It illustrates the key issues prevalent in the development and maintenance lifecycle of mobile application development.
- It alludes to the inherent patterns of mobile app development for distributed and cloud-based mobile applications.

The rest of the paper describes the architecture and individual building blocks of our solution, and is organized as follows. Section 2 provides an overview of the architecture; Section 3 discusses the design considerations for a location service focusing on the Android location service and comparing it with other location providers, describes Android location API components, and the method used to determine a device's current location; Section 4 discusses the design considerations for a cloud-based mobile device notification service focusing on the configuration of Google Cloud Messaging [6] that allows a server to distribute weather notifications to Android devices; Section 5 describes the need for a web service focusing on the configuration of a web server to allow Android devices to post information to the server; Section 6 discusses the improvements needed to the solution after integrating the different pieces; and finally Section 7 presents concluding remarks and lessons learned.

2. System Architecture

This section provides an overview of our mobile application architecture for distributed, decentralized and scalable cloud-based mobile application for real-time weather alert. Figure 1 illustrates the overall architecture of our system. The dissemination pattern reflects how a weather warning alert will be distributed to the relevant users who must be informed of the impending weather event in real-time.

MobileDeli '13, Oct 28, 2013, Indianapolis, IN, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

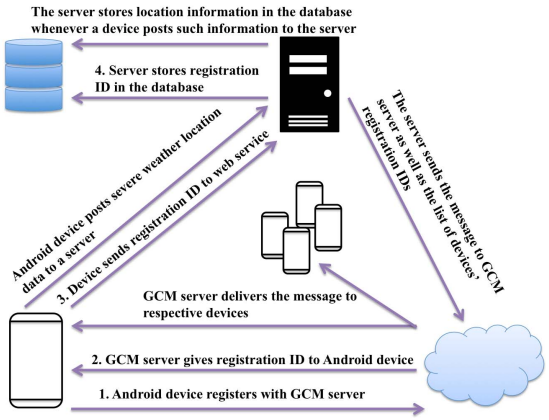

Figure 1. Architecture of Mobile Applications-based Severe Weather Alert System

As shown in the figure, Android devices register with a Google Cloud Messaging (GCM) server (Step 1), which returns an ID to the device (Step 2). The device then sends an ID to a web service (Step 3), which stores an ID in the database (Step 4). These four steps are necessary for the system to work. Android devices could then send geographic coordinates, representing an approximate location of a severe weather to a web service, which will also be stored in a database. The web service, in turn, sends the message containing information about the location of a hazardous weather to the GCM server. The web service also sends a list of registration IDs of the devices to GCM server; this helps to identify the devices that will receive the message. The GCM server then distributes the message to all relevant Android devices.

3. Design Considerations for a Location Service

Android's location service provides access to facilities that can be used to determine a device's location [18]. A location service is needed because our solution must eliminate false alarms and allow devices to send their geographic coordinates to a server. These facilities are discussed below, including an explanation of how we have used these in our solution.

3.1 Choice of Location Providers

Android's location service provides access to location providers that can be used to determine a device's location. A device's geographic location can be determined using either a Global Positioning System (GPS) provider or a Network provider. The characteristics, advantages, and limitations of each are described below.

3.1.1 GPS provider

GPS is useful in determining the current location, but it has some limitations, especially in mobile platforms, such as the time it can take to calculate the current position. To help circumvent some drawbacks of standard GPS, modern mobile devices make use of the following enhancements.

• Assisted GPS (A-GPS): A-GPS uses the mobile network to transmit the GPS information to a mobile device, thus allowing for faster transmission of information from satellites and a faster determination of a device's current geographic location.

• Simultaneous GPS (S-GPS): Devices that use standard GPS may use the same hardware to communicate with GPS satellites and make calls. Hence, only one of these actions can take place at a time. S-GPS adds an additional hardware component to a mobile device, which allows GPS radio and the cellular network radio to operate simultaneously. Considering GPS improvements, some limitations still exist. GPS receivers are unlikely to work indoors and may produce erroneous results in urban areas because GPS signals frequently bounce off of tall buildings.

3.1.2 Network Provider

A network provider can provide location information using wireless network information or cell towers, as described below.

- *Wireless Network Access Points:* Wi-Fi-based location detection works by having a device track what Wi-Fi access points it can detect and the current strength of those signals. The device can then make a query to the Google location service, which provides location data based on Wi-Fi information. One of the main benefits of the Wi-Fi location source is that it allows devices to acquire location information in areas where GPS cannot provide location data. Yet, Wi-Fi networks as a source of location data pose several limitations. First, Wi-Fi networks must be in range. Second, the networks must have a publicly broadcasted service set identifier (SSID) that has not been configured to be ignored by Android. Third, changes to the location of Wi-Fi access points can cause inaccuracies in the location data.
- *Cell IDs:* The cellular network is used in a similar way as Wi-Fi access points to determine a device's location. To function properly, a cellular device must be in contact with a cell tower. Knowing the unique ID of the tower that a device is currently connected to, and possibly the towers that a device was previously connected to, can explain where the device is located. Cell ID-based network provider possesses limitations similar to that of Wi-Fi-based provider, but because the location of cell towers is less likely to change than the location of wireless access points, some complications are removed.

Table 1 compares the discussed location providers in terms of the accuracy and the device's battery consumption of each. We will explain which provider we chose to use in our solution further in the paper.

Location Provider	Battery Consumption	Accuracy
GPS Provider	Consumes more battery power than the Network provider	Provides the most accurate location data
Network Provider	Consumes less battery power than the GPS provider	Provides less accuracy than the GPS provider

Table 1. Comparison of Location Providers and Key Design Considerations

3.2 Android Location API Components

Before we could use the Android location service in our application, we had to become familiar with the available tools. The majority of the classes needed when working with location data are found in the `android.location` package. The four most important classes are `LocationManager`, `LocationProvider`, `Location`, and `Criteria`, and a `LocationListener` interface described below. Figure 2 illustrates the collaboration between these entities.

1. *LocationManager:* This class allows an application to tell a device when it is interested in receiving updated location information and when it no longer wants updates. It also provides information about available location providers, enabled providers, and GPS status information.

2. *LocationProvider:* This class acts as an abstraction for the different sources of location information in Android. Although each provider generates location data differently, they all communicate with an application in the same way and provide similar data to an application.

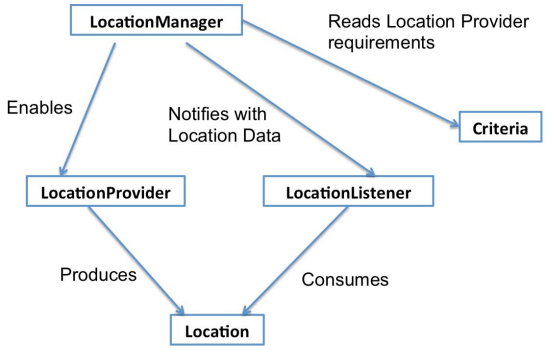

Figure 2. Android Location API Collaboration

3. *Location:* This class encapsulates the actual location data provided to an application from a location provider. The location data includes latitude, longitude, and altitude.
4. *Criteria:* This class queries the `LocationManager` for location providers that contain certain characteristics, such as how accurate is the location data. Using a `Criteria` object is useful for allowing a user to customize the source of location data at run time.

3.3 Determining a Device's Location

The Android location service is most often used in applications that perform the following operations: determining a device's current location, tracking a device's movement, or firing a proximity alert when a device enters or leaves a user-defined area. For the purposes of our solution, the application simply needs to detect a device's current location when a user sees an impending hazardous weather event. Hence we used the appropriate methodology described below.

The location service requires an application to declare its intentions to use it in the Android manifest file. We chose to use the GPS provider in our solution because it provides the most accurate information, and its battery consumption could be lessened by disallowing location updates when they are not needed. Moreover, people who are outdoors are the ones who must be alerted even earlier to get inside to avoid the weather event. Since they are outdoors, the likelihood of them receiving a GPS signal is higher. In order to use GPS location provider, we declared the `ACCESS_FINE_LOCATION` permission in the manifest file, rather than the `ACCESS_COARSE_LOCATION` permission, which provides a more accurate location data. For a different application, such as earthquake alerting, a Wi-Fi-based solution may be preferred.

The implementation of a `LocationListener` interface required the Android application to contain the following four methods:

- The `onLocationChanged (Location location)` method, which is called when a new location is ready for consumption by an application. The parameter to this method is a `Location` object that contains the details about the location. We implemented this method to retrieve the latitude and the longitude of the device's location via the `location.getLatitude()` and `location.getLongitude()` method calls.
- The `onProviderEnabled (String provider)` method that provides a way for an application to be notified when a user enables a location provider.
- The `onProviderDisabled (String provider)` method that provides a way for an application to be notified when a user disables a location provider.
- The `onStatusChanged (String provider, int status, Bundle extras)` method, which is called when a provider either goes offline or comes back online.

We created a class in the main activity of the Android application that implements the `LocationListener` interface. To obtain a reference to the `LocationManager` class, which is the "front door" into the location service, the application makes a call to the `Activity.getSystemService (LOCATION_SERVICE)` in the `onCreate()` method. The application calls the `LocationManager.requestLocationUpdates (String provider, long minTime, float minDistance, LocationListener listener)` method so that it could be provided with location information.

Since we chose to use a GPS provider, we pass `LocationManager.GPS_PROVIDER` as the first parameter to the method. Finally, and just as importantly, the application unregisters the location listener when it no longer needs location updates. This is done with the `LocationManager.removeUpdates(this)` call in the `onPause()` method of the main activity. Forgetting to do so could cause the provider and underlying hardware to remain active, thus wasting battery life.

4. Design Considerations for a Cloud-based Notification Service

Our solution needed a capability that could handle real-time notifications to mobile devices – essentially a publish/subscribe capability – and that too a scalable solution. One choice was for us to develop a solution ourselves, however, that would have resulted in reinventing capabilities that already exist. Hence, we first explored the available solutions in this space. To that end we identified *Google Cloud Messaging (GCM)* [6] as an appropriate choice because interfacing with Android devices was easier. GCM is Google's replacement for its Cloud to Device Messaging (C2DM) protocol [12]. GCM is a service that allows a server to send data to Android devices; hence we incorporated GCM into our solution. Below, we describe the steps taken to configure the GCM capability in the solution.

4.1 Enabling GCM

We created a Google API Project at the Google API console page (`https://code.google.com/apis/console`), where we enabled the GCM service to be used in our project. We also received a project ID, which will be used in the application implementation, and a browser key that will be used in the server implementation. Before we could proceed with the application and server development, GCM helper libraries were installed. This created a gcm directory under `<SDK_ROOT>/extras/google`, which contains the gcm-client, gcm-server, and other subdirectories. These libraries aid in the development of Android and server-side applications. These helper libraries are only one option for creating an Android application that uses GCM. Another alternative is to use GoogleCloudMessaging API, which was added later by Google.

4.2 Writing the Android Client Application

We continued to enhance our existing application that retrieved a device's location as described before with GCM features. The gcm.jar file was copied from the gcm-client/dist directory to the

application's classpath. Several permissions were added to the Android manifest file:

- A custom permission that allows only the given application to receive GCM messages.
- A permission that allows the application to receive GCM messages in general.
- A permission that allows GCM to connect to Google Services.
- A permission asking for GCM to require a Google account.

The `GCMBroadcastReceiver` [4] was also added to the manifest file; it is responsible for handling the Receive and the Registration intents that can be sent by GCM. It should be defined in the manifest, rather than programmatically, so that the intents can be received even if the application is not running. The `GCMIntentService`, which is an application-provided subclass of `GCMBaseIntentService`, was also declared in the manifest. This service will be called by the `GCMBroadcastReceiver` class, which is provided by the GCM library.

In the following we describe the four classes of the application. Figure 3 illustrates the collaboration between these four classes.

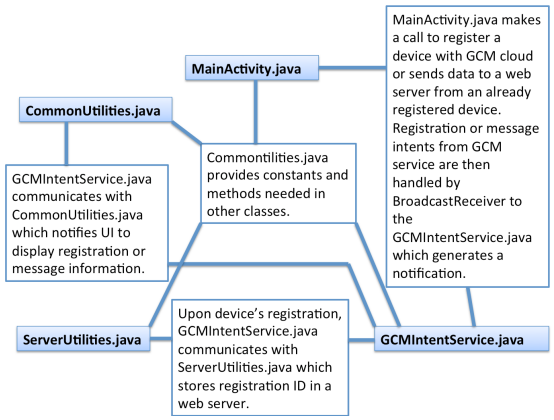

Figure 3. GCM Class Collaboration

1. *MainActivity.java:* This class stores the URL of a web service as a string (the configuration of a web service will be described in later sections). The registration of a device with GCM is usually done during the onCreate() method, so we implemented the appropriate call there: the `registerClient()` method checks the current device via `GCMRegistrar(Context c)` call, the manifest for the appropriate rights via the `GCMRegistrar.checkManifest(Context c)` call, and then receives a registration ID from the GCM cloud via the `GCMRegistrar.getRegistrationId(Context c)` call. If there is no registration ID, meaning that the device has not yet been registered with GCM, the GCMRegistrar will register the device for the project by calling `GCMRegistrar.register (Context c, string webServiceURL)` and return the ID by calling the `getRegistrationId()` method, described above. The registration ID would then be automatically sent to the web server, so that the server would know which devices to communicate the messages to (described further).
2. *GCMIntentService.java:* This class is responsible for handling GCM messages. It extends `GCMBaseIntentService` and overrides the following important methods.
   (a) `onRegistered(Context c, String rID)`: This method is called when a device registers with GCM. It calls a method in the ServerUtilities class, `ServerUtilities.register(Context c, string rID)`, which will send the device's ID to the web service.
   (b) `onUnregistered(Context c, String rID)`: This method is called after a device has been unregistered from GCM.
   (c) `onMessage(Context c, Intent i)`: This method is called when a new message is received from a web service. We implemented the method to retrieve the message from the intent, passed as a parameter, and issue a notification to inform the user about the received message. Additionally, we update the user interface with the new information.
   (d) `onError(Context c, String e)`: This method is called when a device tries to register or unregister, but GCM returns an error. We simply print the message to a log file.
3. *ServerUtilities.java:* This class is intended to communicate with the web server when a device registers or unregisters with GCM. Therefore, it contains the following methods.
   (a) `post (String URL, Map<String, String> params)`: This method posts a device's registration ID to the web server by doing a HTTP POST request.
   (b) `register(final Context c, final String rID)`: This method sends a device's registration ID to the web server by calling the above described `post()` method and passing the URL of a server that handles the storage of the ID, and the map of name/value pairs representing parameters to be posted to the server. In our case, the map contains the "regID" name that serves as a tag for the actual registration ID value. If a registration ID is sent successfully, the method calls the `GCMRegistrar.setRegisteredOnServer(c, true)` to assert that an ID was sent.
   (c) `unregister(final Context c, final String rID)`: This method calls the `post()` method, passing the URL of a server that handles the removal of the ID from the server.
4. *CommonUtilities.java:* This class is a helper class that provides constants such as server URL, Sender ID, intent constant to display a message on a screen, and the name for the message to be displayed used as a tag in intent's extra. It also defines a method that notifies UI to display a message. These constants and the method are needed in other classes of the application.

5. Design Considerations for a Web Service

Note that GCM is just a messaging capability that can handle large-scale notifications to mobile devices. However, it cannot by itself understand the semantics of a weather alert nor can it know whom to notify. This and any other information, in case of applications with different purposes, must be provided through external means, and that is why GCM requires a 3rd party web server. To that end we developed a cloud-based server that can track all the registered devices and current alerts.

In order to write a server-side application hosted in the cloud for our solution, we created an Amazon EC2 Instance using Amazon Web Services [1, 16]. The Amazon EC2 Instance runs Linux 3.4. We installed Apache HTTP Server [2, 15] to create the required by GCM 3rd party web server, MySQL [7, 14] to have a relational database management system, and PHP [13] to run server-side scripts to interact with the MySQL database. The MySQL database will help our server to track and use important for our solution information, such as the registered devices' IDs and current alerts.

5.1 MySQL Database

We created a ”gcm” database that contains two tables: ”data” and ”gcm_users”. The ”gcm_users” table stores GCM registration IDs of registered devices and the timestamp indicating the time when those IDs were saved on the server. The ”data” table stores the approximate latitude and longitude of a severe weather event, as well as the registration ID of a device that posted the location information to a server, and the timestamp when the data was saved.

5.2 Server-side Programming

We created several PHP files to handle a device's registration on a server, an automatic receiving of an incoming notification, and a dissemination of the message to other registered devices. The purpose and implementation of each class defined in the PHP file is described below.

- *db_connect.php:* This file defines a `DB_Connect` class that contains `connect()` and `close()` functions to open a connection to a database and close it, respectively. The `connect()` function uses a `mysql_connect(server, user, password)` PHP function to connect to a server, and the `mysql_select_db-(database)` to select the corresponding database. It then returns a database handler to the caller. The `close()` function calls `mysql_close()` PHP function to close the database connection.
- *db_functions.php:* This file defines a `DB_Functions` class that implements functions, which perform database queries. The constructor of the class contains code to connect to a database using functions from the file `db_connect.php`. The `storeUser($regID)` function takes a device's GCM registration ID as a parameter and uses the `mysql_query(query)` PHP function to insert the ID into the “gcm_users” database table. The `storeData($latitude, $longitude, $regID)` function takes the location of a severe weather event in terms of latitude and longitude, and a device's registration ID as parameters and inserts the given information into the “data” table of the database along with a timestamp.
- *GCM.php:* This file has a class `GCM` that takes care of sending push notifications to Android devices. It contains the `send_notification($registration_ids, $message)` function that handles the transmission of a message. It defines a URL of a GCM server where the message should be sent, an associative array 'fields', an abstract data type composed of a collection of unique key/value pairs, containing registration IDs and the message, and an associative array 'headers' containing the Google API Key and the JSON format content-type specification that will be used for the HTTP header.

  The function then initializes the cURL via `curl_init()` PHP function. cURL is a library that lets one to make HTTP requests in PHP [8]. It is important in our solution since we are trying to make a HTTP request to the GCM server. The function then sets the URL via the `curl_setopt(curl instance, CURLOPT_URL, url)` PHP function. It then calls `curl_setopt(curl instance, CURLOPT_POST, true)` to define a regular HTTP POST to the GCM server. The `curl_setopt(curl instance, CURLOPT_HTTPHEADER, headers)` sets an array of HTTP header fields. The `curl_setopt(curl instance, CURLOPT_RETURNTRANSFER, true)` function allows the transfer to return as a string of the return value of `curl_exec()` instead of outputting it out directly. The `curl_setop(curl instance, CURLOPT_POSTFIELDS, json_encode(fields))` function sets the data, formatted as JSON, to post in an HTTP POST operation to a GCM server. The POST request is finally executed using the `curl_exec(curl instance)`, and the connection is closed via `curl_close(curl instance)`.
- *register.php:* This file handles the registration of a device on a server. As mentioned earlier, upon a registration with a GCM server, a device receives a GCM registration ID, which is then sent automatically to our server. The code in `register.php` checks if the POST request to our server contains the registration ID via `isset($_POST[''regID''])`, where “regID” is the name that distinguishes the registration ID value. We used this name as a tag in the Android app when we created a map of name/value pairs to be used as a parameter in the POST request to our server, where the name was the “regID” and the value was the actual registration ID. Therefore, if the POST request contains the registration ID, 'register.php' then stores this ID in the 'gcm_users' database table via the `storeUser(regID)` function, defined in the `db_functions.php` class. It then creates an array containing the registration ID, and an associative array containing a message, signifying a successful registration. These arrays are the passed to the `send_notification(registration_IDs, message)` function of the GCM.php class, which notifies the device of a successful registration.
- *store_location.php:* This file contains PHP code that receives a latitude, a longitude, and a registration ID data in a similar way the code in the register.php receives a registration ID. It then stores information in the 'data' database table using `storeData(latitude, longitude, regID)` function of the `db_functions.php` class. It then calls the `prepare_for_sending(latitude, longitude)` function of the `send_message.php` file (described below) to distribute severe weather location to registered Android devices.
- *send_message.php:* This file defines a class `Send` which contains the `prepare_for_sending(latitude, longitude)` function. The function performs a connection to a database and executes a query to fetch all devices' registration IDs from the 'gcm_users' table. The IDs are stored in an array. The function then defines an associative array containing a message, and finally calls the `send_notification(regIDs, message)` function of GCM.php to distribute the message to the registered devices.

6. System Improvements on the Mobile Device

Our earlier discussion focused on server-side and cloud-based capabilities that we introduced in our solution. We were required to make a few enhancements on the mobile devices. When the server we built was ready to both send and receive data, the Android application still required the implementation of how the location data would be sent to the server, which will be described shortly. We also added several improvements to the Android application, which also will be described in this section.

6.1 Posting Location Information to the Web Service

We allowed a user to send the device's geographic coordinates by clicking a button on the device's screen. For example, when a user sees an approaching severe weather event, he/she can start the application on a smartphone, and click “Send My Coordinates” button. The `onCreate()` method of the main activity monitors when the button will be clicked via the `View.OnClickListener`, which implements the `onClick(View v)` method that calls the `onSendButtonClicked()` method. This method retrieves the current location of a device as we discussed before, constructs an object that will post the location data to a server in a background thread, and calls the `execute()` method of that class.

The class is a private class in the main activity that handles POST requests to the web service in the background. It extends `AsyncTask` [3] which encapsulates the creation of Threads and Handlers. It defines an array of NameValuePairs to hold parameters to be posted to a server, such as latitude and longitude. Any class that extends AsyncTask needs to implement the three methods, described below.

- `onPreExecute()`: This method creates and shows a progress dialog to the user while a lengthy POST operation is being executed.
- `doInBackground(String ...urls)`: This method performs a lengthy operation, such as HTTP POST, in the background thread. We generalized the method to perform both POST and GET tasks.
- `onPostExecute(String response)`: This method is called after a lengthy operation has completed. It dismisses the progress dialog and synchronizes itself with the user interface thread.

6.2 Detecting Internet Connection

We created a `ConnectionDetector.java` class, which serves to detect the Internet connection status via the Android's *ConnectivityManager* [5]. For this feature to work, a permission was added to the Android manifest file that allows an application to detect the Internet status. We also added an `AlertDialogManager.java` class which simply shows an alert dialog, such as to inform a user when there is no Internet connection.

The `onCreate()` method of the `MainActivity.java` calls the `isConnectingToInternet()` method of the `ConnectionDetector.java` class which returns a boolean result. If the Internet connection is not detected, the `showAlertDialog()` method of the `AlertDialogManager.java` class is called in response, asking a user to connect a device to the Internet.

6.3 Waking up a Device on Receiving a Notification

We created a `WakeLocker.java` class, which serves to wake up a device on receiving a new notification if a device is sleeping [17]. A permission that allows the application to wake up a device was also added to the Android manifest file. This is important, because a user must receive a notification about approaching weather event even when he or she is not using a device. Therefore, if a message is received when a device is sleeping, the Broadcast Receiver defined in the `MainActivity.java` acquires the wake lock in the main activity via `acquire()` call to a `WakeLocker.java` class; this wakes up the phone to show the message to a user. When a user has been notified, the `release()` method call releases a wake lock.

Whether to create a Wake Lock or not is a key design consideration. It can have a dramatic impact on a device's battery life. If an application needs Wake Locks, it is a good design practice to create them only when necessary, and holding them for short period of time only.

6.4 Adding a Notification Sound and Device Vibration

We allowed our application to play a custom notification sound, and also to vibrate the device when a message is received. This design consideration was made to better alert users when an important notification is received.

6.5 Eliminating False Alarms

As we discussed before, one of the goals of the solution was to disallow a user to be notified of a severe weather event if the user is not located in the area of hazardous weather even though the device may be registered. We described how the `onMessage()` method in the `GCMIntentService.java` class of the Android application is called when a new message is received from the web service. This method retrieves the message from a parameter passed as Intent and generates a notification to the user.

To eliminate false alarms, the method needs to get a device's present location as discussed previously and calculate its proximity to the location where the hazardous weather event might occur next. We still have to work on this improvement. The application would need to define a radius, or even allow a user to define a radius, around a possible hazardous weather location, such that if a user's device is located within the radius, a user would receive a notification, and a user would not receive a notification if his/her device's location is outside the radius. This work is part of our future work agenda. A key consideration is to understand the dynamics of the event (since the direction of the weather event may change).

7. Conclusions and Lessons Learned

This paper described the design of a decentralized, scalable, and easy-to-use severe weather warning system based on publish/subscribe communications and implemented as a mobile application on Android devices, and cloud-based servers. This project incurred a steep learning curve and integration challenges, particularly for two undergraduate students who worked on this problem as an independent study/summer research projects. The insights we gained thus far are qualitative.

The project helped us to learn the following lessons and addressing current limitations forms part of our future work:

- *GCM technology eliminates extra work:* We learned that the GCM technology makes it possible to eliminate the requirement for an application to query the server for the content because the server itself initializes the distribution of data to relevant devices. Thus, application developers can use this insight to avoid duplicating work.
- *Patterns of reuse:* Our solution could be easily adapted to work in similar situations with minimal changes. In other words the basic communication and integration patterns from our work can be applied to a variety of related application problems. However, some configuration changes are needed. For example, currently it uses GPS provider, which gives more accurate alerts to people who are outdoors to make them go inside to avoid severe weather event. For an earthquake system, however, it would be more efficient to use the Wi-Fi provider instead, which would work better to alert people who are inside for them to get outside. Similar other scenarios, such as app development in alerting for flash floods, is an area we are working on to get additional insights in documenting the patterns of reuse from our work.
- *Notification alternatives:* Our solution uses Google Cloud Messaging as a means for sending push notifications to Android devices. There are other alternatives possible. For example, one could use Urban Airship [10], which itself uses GCM to support push notifications. It allows sending notifications both from the Google side and the Urban Airship side. Another alternative could be Xtify [11], which offers APIs that allow developers to integrate push notifications within existing applications. It also provides Location Services that allow location-triggered messages. Yet another alternative is Parse [9]. Its library provides push notifications by running a background service that keeps an Internet connection to the Parse Cloud.
- *Dealing with accidental complexities using model-driven engineering:* We created our solution using an approach that involves significant accidental complexity, and hence manual development is error-prone. For example, when our applica-

tion calls the `LocationManager.requestLocationUpdates (String provider, long minTime, float minDistance, LocationListener listener)` method, we pass `LocationManager.GPS_PROVIDER` as the first parameter to the method to ensure that the application uses GPS provider, and not any other one. Such a programmatic tight coupling that is application-dependent is the responsibility of a programmer. Any mistake will lead to the solution becoming unreliable. Many such instances may exist in a mobile application design, which constrains reuse and becomes a maintenance nightmare.

To overcome this problem requires configurability and automation. We concluded that it might be beneficial to automate some of the code generation, possibly using model-driven engineering. We could, for example, use generators that analyze models and synthesize source code, thus avoiding software development approaches that are tedious and error prone. Another situation in which model-driven engineering would be beneficial is the declaration of permissions in the Android manifest file. Permissions are specific to a particular application, and the developer must be certain to include them for the application to operate properly. This could be another opportunity for automation, where specific permissions would be auto-generated for a particular application requirements. Yet another error-prone situation is the definition of the GCMBroadcastReceiver in the manifest file rather than programmatically handling intents when the application is not running.

- *Integration challenges and dealing with evolution:* We also realized that usage of 3rd party tools, such as Google Cloud Messaging and Amazon Web Services, implies that we must cater to their requirements and also changes, if any, they make. As an example, before GCM, Google provided the Android Cloud to Device Messaging (C2DM) to support push notifications on Android devices. Now C2DM is deprecated, and developers using it are encouraged to move to GCM. Thus, there is always a maintenance challenge and also a threat of vendor lock-in. Model-driven engineering may be useful to abstract out the technology- and vendor-specific details and use generative mechanisms to synthesize glue code thereby avoiding vendor lock-in.
- *Supporting an app from app store:* Our Android client application can be made available as a downloadable application from the App Store, which will automatically connect to the GCM server, and our Amazon cloud server on its use. We have yet to analyze how much our application adheres to the compliance policies of Android. We were able to test the system with a server and only a few Android devices. Experiments of large-scale setup are needed to understand the scalability of the design. Moreover, handling the dynamics of the weather event is another consideration we need to handle. Finally, applicability of the design to other societal challenges needs to be studied.

Acknowledgments

We would like to thank the generous support of the EECS Department in supporting this work. Part of the work was also supported by NSF CAREER award 0845789 REU grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Amazon Web Services. `http://aws.amazon.com/`. Accessed Aug 2013.

[2] Apache HTTP Server Project. `http://httpd.apache.org/`. Accessed Aug 2013.

[3] Android AsyncTask. `http://developer.android.com/reference/android/os/AsyncTask.html`. Accessed Aug 2013.

[4] Android BroadcastReceiver. `http://developer.android.com/reference/android/content/BroadcastReceiver.html`. Accessed Aug 2013.

[5] Android Connectivity Manager. `http://developer.android.com/reference/android/net/ConnectivityManager.html`. Accessed Aug 2013.

[6] Cloud Messaging for Android. `http://developer.android.com/google/gcm/index.html`. Accessed Aug 2013.

[7] MySQL: The World's Most Popular Open Source Database. `http://www.mysql.com/`. Accessed Aug 2013.

[8] PHP Client URL Library. `http://php.net/manual/en/book.curl.php`. Accessed Aug 2013.

[9] Android Push Notifications. `https://www.parse.com/tutorials/android-push-notifications`. Accessed Aug 2013.

[10] Android: Getting Started with Push. `http://docs.urbanairship.com/build/android.html`. Accessed Aug 2013.

[11] xtify: Mobile Customer Engagement. `http://www.xtify.com/`. Accessed Aug 2013.

[12] Android Cloud to Device Messaging Framework. `https://developers.google.com/android/c2dm/`. Accessed Aug 2013.

[13] M. Achour, F. Betz, A. Dovgal, N. Lopes, H. Magnusson, G. Richter, D. Seguy, and J. Vrana. PHP: Hypetext Processor. `http://www.php.net/manual/en/index.php`. Accessed Aug 2013.

[14] P. DuBois. *MySQL*. Pearson Education, 2008.

[15] R. T. Fielding and G. Kaiser. The apache http server project. *Internet Computing, IEEE*, 1(4):88–90, 1997.

[16] R. S. Huckman, G. P. Pisano, and L. Kind. Amazon Web Services. *Harvard Business School Case*, (609-048), 2008.

[17] R. Meier. *Professional Android 4 Application Development*. John Wiley & Sons, 2012.

[18] G. Milette and A. Stroud. *Professional Android Sensor Programming*. John Wiley and Sons, 2012.