Virginia Tech AED Locator Android Application

Mark Conover
conoverm@vt.edu

Charlie Cook
cmcook@vt.edu

Craig Estep
craigy@vt.edu

Robert Lyerly
rlyerly@vt.edu

ABSTRACT
In this paper, we describe the development of an Android application for finding Automated External Defibrillators (AEDs) on the Virginia Tech campus.

1. INTRODUCTION
Over 446,000 people in the United States fall victim to sudden cardiac arrest annually. A person experiences cardiac arrest when they have no pulse because their heart has suddenly stopped beating. As minutes pass, heart muscle dies and the body's vital organs are deprived of oxygen rich blood. An AED gives an electrical shock to restart the heart within 2 to 3 minutes and gives the person a 90 percent greater chance of surviving than someone who does not receive the shock. In order to increase the survival chances of a person within the Virginia Tech campus and to decrease the annual victim total, an Android mobile application to help people find the nearest AED on campus was developed.

2. ACKNOWLEDGMENTS
Our thanks to VT Rescue for allowing us to develop their idea.

3. HOW IT WORKS
The Virginia Tech AED Locator Android application is made for all devices that have an Android operating system. The components used to run the application are a Microsoft Windows Azure server with database, SQLite database, GPS and Network provider, and Google Maps API.

4. WINDOWS AZURE
For our project, we were given a 3-month license to use Windows Azure, Microsoft's solution to Cloud Computing. Windows Azure is a platform for hosting web services and applications, handling domain name services and allowing flexible scaling [1]. After registering for our license, management of the service was provided through a clean but robust and full-featured web interface (note that this web interface utilizes Microsoft Silverlight, so compatibility is limited on non-Windows platforms). It allowed us to register services (web applications and relational databases), deploy applications in a staging or production environment, and scale our applications to fit demand.

Using Windows Azure was fairly simple; there are many existing tools to help with development and deployment. Installing the Windows Azure SDK provides several tools which are necessary to create an Azure application. The first is the packaging tool which builds a deployable-package from your web application code. It creates all of the necessary files to deploy your application to the cloud. The second is a set of emulators used for testing your web application locally. These tools simulate the Azure OS on your computer so you can test without having to deploy the application to the cloud after every change, a very slow process. Since the emulator runs a virtual machine containing the Azure OS, the way your application behaves locally mirrors the way your application runs on the cloud.

4.1 Azure Server
The server was developed in the SpringSouce framework. It comes packaged in a WAR file along with the java run-time environment. This allows the server to be easily deployed on a with variety of platforms. For our service we deployed it in a Tomcat instance running on the Microsoft Azure cloud. The server is responsible for maintaining the master list of AEDs and determining which AED records have been changed when a client asks for an update, this is done by attaching a timestamp to each record and updating the timestamp every time that record is changed, the client simply provides the last time it updated and all new or changed records since that time are sent back to the client. AED records can also be created, modified and destroyed by authenticated users. The authentication is achieved by the authentication manager in the Spring framework and is tracked using session ids so that once a client has authenticated itself it may perform the privileged actions for a set amount of time before having to renew its session.

4.1.1 *Apache Tomcat*
Since we were designing the AED locator app for Android phones, we did not have the close integration with Windows Azure that developers for Windows Phone have. Also, Android phone apps are written in Java, eliminating some of the tools that come with the Windows Phone SDK that uses C#. We decided to create a more flexible cloud interface that could run on any platform. We chose to deploy our web application using the Apache Tomcat web server, a mature, robust web server that runs on any system that has a Java Virtual Machine (JVM) installed [2]. It also provides easy web application deployment features that interacted nicely with Spring. Using Apache Tomcat, the web application portion of our project can be run on any webserver, including other cloud computing platforms and pre-existing servers.

4.1.2 *Server-side Improvements*
There are several possible improvements that would enhance our system further. We could make a web interface for our database so that VT Rescue team members could log into a website and easily add, edit or delete one or more AEDs at a time. As of right now, the only interface for doing this is on the Android app which only lets you add or edit one device at a time. Another future improvement would be to possibly offload some of the computation to the server. One of the nice things about our application is its ability to run even without an internet connection. However, if the app is connected to the internet, utilizing the resources of the server could potentially increase the efficiency of the application in terms of computational overhead and power consumption.

4.2 Azure Database
The server's list of AEDs is persisted using the Hibernate framework. Hibernate provides a single interface for the persistence and querying of objects to the server but allows the actual instantiation of the database to vary. This allows the

database to be able to be changed, or swapped out for another type of database, without affecting the server. Additionally, the Hibernate framework can analyse the object's code can update the tables of the database if new fields are necessary. This provides for easier maintenance and updates to the server code without having to update the database manually.

4.3 Windows Azure Plugin for Eclipse

Another tool that we utilized was the Windows Azure Plugin for Eclipse with Java (since Spring is built on top of Eclipse, the plugin was compatible with Spring as well) [3]. It integrates many of the tools of the Azure SDK with Eclipse – you can create Windows Azure projects, build the projects for the emulator or for deployment to the cloud, adjust minute details of your web application (such as which port on which to accept connections) and it provides scripts that handle setting up the emulator to simulate your web applications. The tool allowed easy project management; the web application could be edited in Spring, then dropped into an Azure project and simulated.

5. SQLITE DATABASE

The SQLite database is for storing a list of Virginia Tech's campus buildings and AEDs with their corresponding GPS coordinates. The application queries the SQLite database to find the Virginia Tech building that is closest to the device and finally the AEDs that are reside in that building. The SQLite database is updated either daily or weekly, depending on the set user preference, by syncing with the Azure server's database. In order to increase a device's battery life, all operations for determining new AEDs and Virginia Tech buildings that need to be added to a device's SQLite database is done on the Azure server. A device simply sends an HTTP request with a timestamp of the last time the device sync'd with the Azure server's database. Thus, the Azure server is able to determine the new AEDs and Virginia Tech buildings for that device and return them in XML format within an HTTP response.

6. MAP

The most fleshed-out method for finding AEDs is also the most useful for navigating to one. It provides a visual way to find an AED as well as tools for locating and navigating to a chosen AED.

There are actually multiple cloud services working behind the scenes that allow us to hook into them. The first is the Google Maps API, specifically for Android, and the second is the Directions API, which we use to get walking directions from the user's location to a given AED.

6.1 Google Maps API

Because our app runs on the Android platform, we have the opportunity to utilize the Google Maps API. It is very simple to enable Google APIs for an Android project, with a little more setup required to get the map working.

To use maps, an API Key is required. It is free from Google and is very easy to acquire. However, we ran into the problem that each developer on our team (four of us) needed a separate key, as they are tied with the local Android SDK. There are ways to get a shared key for a larger team but because our team is so small it was simply a minor annoyance.

6.1.1 Features

Like the web interface, the Google Maps API for android uses tiles and different zoom levels to display a map to users. From there, using Overlays or custom rendering, as a developer you can customize the display and contents of the map.

It supports both standard maps and satellite view, which is very useful for us because it could give the user a better perspective about their location if location services were not available.

6.1.1.1 MyLocationOverlay

A useful feature in the library is the overlay provided for displaying the user location on the map, using the Android Location API which hooks into GPS, IP tagged location using WiFi, and network tower triangulation.

For our app we customized this class with a compass direction indicator. The way we accomplish this is to request updates from the compass, then change the current icon shown on the map using eighteen icons from the open source My Tracks project which are arrows pointing in those various directions.

6.1.1.2 Custom Location

The MyLocationOverlay is only useful if location services are operating and accurate. This would not be the case if the user was inside with only GPS enabled and no other location services. The GPS chip would try to get a fix but it is very unlikely that it would be able to without a clear view of the sky.

Our solution, which is similar to our list based method for choosing an AED by choosing a building, is to allow the user to manually enter a location on the map. This is done either by long-pressing on the map and using the context menu, or through the options menu which is opened using a key. These are both standard methods for selecting a choice like this, with the options menu more visible.

Once the user chooses a location, the Activity will automatically load AEDs near that point. It will basically treat it as the users location from that point on, and they can navigate to AEDs as they normally would from an automatic location. If they want they can also switch to automatic again and try to pick up GPS or other location signals.

6.1.2 Limitations

Compared with the Google Maps app, the feature support in the public Maps API in MapView is very limited. Some examples are it does not support rotation, which would be useful for a navigation mode.

It also does not support many of the layers that the full app does, such as traffic, terrain, transit, and others.

A very new feature was introduced to the main app as well. Indoor maps would be an amazing feature in our program, as at the moment we can only guide users to the building itself, and provide a textual description of the location of the defibrillator inside the building. With indoor maps we could show them a visual indication. Unfortunately it will take a long time for this feature to trickle down into the API, if it comes at all.

6.1.3 Conclusion

The Google Maps API makes a map display possible, and abstracts away much of the work, such as downloading tiles of the

appropriate zoom level, rendering, and even mapping latitude and longitude points to allow us to do our own custom drawing.

6.2 Directions API

Once the user has determined their location and the nearest AED they need a way to navigate to that AED. We help them do this by providing directions using the Google Directions API. The way this API is used is by forming an HTTP request, and then receiving XML or JSON output containing what amounts to a list of points that the user will travel.

To go into more detail, a response is layered into different aspects. These are Routes, Legs, Steps, and finally Polylines. Here is a description of the various aspects and how we utilize and present them.

6.2.1 Routes and Legs

A route represents a way to get from the source to the destination, and the response can contain multiple routes. We do not want to overwhelm the user with options, though, so we simply ignore all but the first route.

Legs represent a section of a route, and so the sum of all the legs is the entire journey. For simplicity, we simply combine the legs by iterating over all of their components, steps.

We use a custom Overlay to draw the route on the map.

6.2.2 Steps

A step represents a latitude and longitude along the way, with an instruction on how to proceed to the next step. It contains text as well as a numerical value to describe how far away the next step is.

Using the custom Overlay, we draw a circle at each step point.

6.2.3 Polyline points

A polyline is an encoded set of points. Because there is no utility in the Google Maps API for decoding a polyline into latitude and longitude points, we used a class from a third party to do this.

Once the points are decoded, we draw a line connecting each polyline point in the entire route.

6.2.4 List format

In addition to the graphical display of the directions, we provide an options menu item that lets the user see a list of the steps, as well as on overall summary of the route. If they click on an item in the list it will immediately take them to the map.

6.2.5 Conclusion

Because finding an AED is so time-critical, and we assume that they will be close enough to travel on foot, we give the user walking directions. It is doubtful that other types of directions would be useful, because we are dealing with such a time-critical situation. It would simply take too much time to get in any sort of vehicle to find an AED.

7. PERFORMANCE

Important to any software is performance. For mobile apps in particular, perceived performance is crucial. If the interface feels slow or unresponsive for a period of time, then users will simply uninstall the application and replace it with something better.

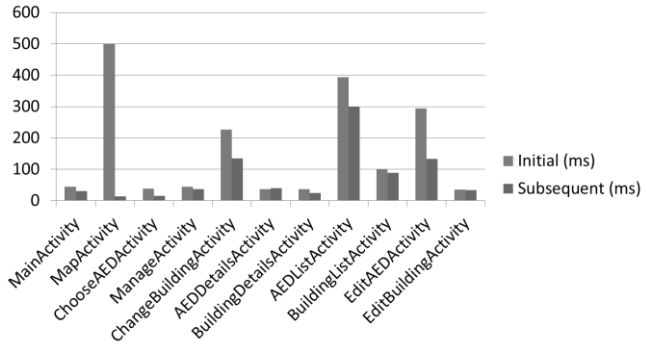
The Android Developers site has a page titled Designing for Performance. They give many tips and examples of how to improve performance. An interesting fact they state is that users will be put off by a 100ms-200ms delay when performing an action. Our goal, therefore, is to reduce the perceived lag to below this threshold wherever possible.

7.1 Profiling onCreate

While it is possible to try to 'eyeball' performance, in order to start optimizing it is important to quantify it first. Switching Activities turns out to be a common area for performance degradation in our app, so we put simple clock measures at the beginning and end of the onCreate method, which is where an Activity is created before being shown to the user.

The profiling showed that there is usually quite a difference between starting an Activity for the first time, and starting it subsequently from then on. In some cases the difference is quite drastic.

Here is the graph of our findings.



7.2 Problem Areas

There are some 'usual suspects' that cause these performance issues. They are database reads and writes, as well as network communication.

7.2.1 Database

Loading items from our SQLite database is time consuming. On average, it takes over 200ms to load AEDs or Buildings, which is well outside the target time.

One reason behind this is because the database is stored on disk. However, this does not fully explain it, because a query for the number of items in the database can be run at around 40ms. The queries for the objects take so long because usually they are done based on the location of the user, so a formula is applied to the SQLite query.

7.2.2 Network

Network activity, for example downloading directions, is very slow. It can take seconds, or at the very least over half a second.

The benefits of using cloud services are met with the downside of slow network bandwidth and response times.

7.3 Solutions

Because this is such a common issue, there are a number of common approaches to solving it. Here are two aspects of the problem and the solutions available.

7.3.1 Threading

Android applications use a 'main' thread to draw the user interface. In order to run a time consuming operation without affecting the user interface, it must be run in a separate thread with updates posted to the main UI thread.

The first way this can be done is using the standard Java Thread object, and using callback methods to post updates to the user interface controls. This is the approach we went with in our app.

A more advanced method is to use Android's AsyncTask class, which has methods for the computation, updates, and result.

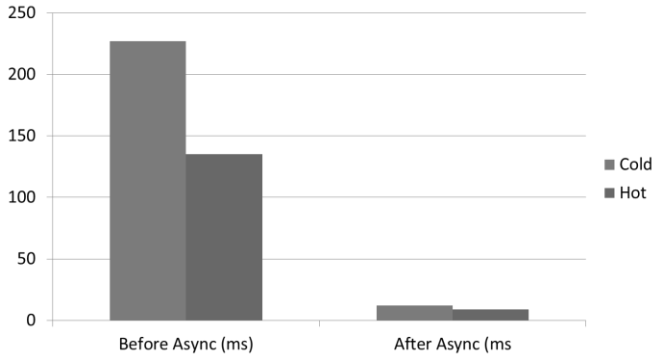
7.3.2 User Feedback

Running operations in a separate thread is not enough, however. The user must be kept informed about what is going on.

In most cases, we simply display a spinner and some text describing what is going on. Other times we display a numerical count to estimate how far we are. In both of these cases we use the ProgressBar widget and post our updates to it from a Thread, or remove it when the operation finishes.

7.4 Effect of solutions

The effect of using a different thread to perform computation is shown below for the ChangeBuildingActivity, which involves loading items from the database.



7.5 Library loading

The map Activity is very slow to start for the first time, probably due to the loading of libraries when the map is created for the first time. Unfortunately there does not seem to be much we can do to alleviate this, as the View itself must be inflated on the UI thread, and so there is no way to move this operation to a separate Thread.

7.6 Conclusion

Android provides many built-in and standardized ways to deal with long running processes. We were able to take advantage in order to reduce perceived delays caused by long running operations to negligible amounts.

8. REFERENCES

[1] Apache Tomcat – an open source software implementation of the Java Servlet and JavaServer Pages technologies. Copyright © 1999-2011, the Apache Software Foundation. http://tomcat.apache.org/

[2] "Community Material." HIBERNATE. RedHat, n.d. Web. 12 Dec 2011. <http://www.hibernate.org/docs>.

[3] "SpringSource Documentation." SpringSource. VMware, n.d. Web. 12 Dec 2011. <http://static.springsource.com/projects/documentation/index.html>.

[4] "University Public Access AED Program." Virginia Tech Rescue Squad. VT Rescue Squad. Web. 2 Dec. 2011. <http://www.rescue.vt.edu/aed.php>.

[5] Windows Azure. Copyright © 2011 Microsoft. http://www.windowsazure.com/en-us/

[6] Windows Azure Plugin for Eclipse with Java. Copyright © 2009 Soyatec. http://www.windowsazure4e.org/