VTGemini: Universal iOS Application for Guided Emergency Response and Notification for the Virginia Tech Community

Kyle Lynn Schutt

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Osman Balci, Chair
James D. Arthur
Eli Tilevich

April 23, 2013
Blacksburg, Virginia

Keywords: Emergency management, iOS mobile software engineering, emergency notification, emergency guidance
VTGemini: Universal iOS Application for Guided Emergency Response and Notification for the Virginia Tech Community

Kyle Lynn Schutt

Abstract

The ubiquitous use of mobile devices and smartphones in the United States presents an interesting opportunity for application developers with respect to emergency management. Software engineers from the federal government to individuals have recognized the unique prospect of utilizing always-connected devices to assist in emergency notification, preparedness, and response. The federal government has instituted and ratified multiple acts and mandates with respect to mobile communications during a crisis such as the Commercial Mobile Alert System. Likewise, individual organizations and developers have created mobile applications that access weather alerts from the National Weather Service. Many of these applications utilize push notification architectures to notify users and stakeholders about impending emergency situations. While most of these applications are geared towards a national audience, there are a few that are highly granular with a focus on the local community. This thesis presents a universal iOS application running on all iOS mobile devices: iPhone, iPad, iPad Mini, and iPod Touch for the Virginia Tech community. The application is highly granular with respect to emergency response guidance and notification by providing clear, concise, and supportive information to citizens during a crisis. Additionally, the application provides another medium of delivery for the Office of Emergency Management at Virginia Tech to potentially mitigate the extent of collateral damage and secondary incidents while saving lives.
Acknowledgments

First off, I would like to acknowledge and thank Officer Geof Allen of the Virginia Tech Police Department for proposing the original idea for a mobile application that guides citizen in the proper response to an incident. Without his idea and subsequent communications with Dr. Osman Balci, this application would not have been possible.

With that, I would like to thank Dr. Osman Balci for his support during the research. He provided great advice throughout my studies at Virginia Tech, both professionally and personally.

I would like to acknowledge Michael Mulhare of the Office of Emergency Management (OEM) for his belief, support, and encouragement for my research and development of this application. He proved to be an invaluable asset to the project and helped us achieve our goals. Additionally, I would like to thank Kendall Woodard for her work on the user interface with respect to the icons, images, and layouts. Her work single-handedly brought our application and the OEM website together visually.

I would like to thank Layla Wang, who provided code and guidance for the expanding table view rows, and Michael Enriquez for developing the sliding view controller. Furthermore, I would like to acknowledge Carl Harris of Communications Network Services (CNS) for feedback on the VT Alerts integration, and for developing a static library that allowed our application to register and receive emergency notifications. I would also like to acknowledge Michael Irwin for his assistance in creating the VT Alerts static library and server side applications. Additionally, I would like to thank Peter Sforza for his support as a committee member at-large.

Finally, I would like to thank Serdar Aslan and Vincent Ngo for software testing and providing valuable feedback.
# Table of Contents

Abstract.............................................................................................................................. ii

Acknowledgments............................................................................................................. iii

List of Figures ...................................................................................................................... vi

Chapter 1: Introduction...................................................................................................... 1
  1.1 Problem Definition ...................................................................................................... 1
  1.2 Objectives of the Software Application ..................................................................... 1
  1.3 Importance of the App .............................................................................................. 2
  1.4 Thesis Overview ......................................................................................................... 2

Chapter 2: Background...................................................................................................... 3
  2.1 Virginia Tech Police Department .............................................................................. 3
  2.2 Office of Emergency Management at Virginia Tech ................................................ 3
  2.3 Communications Network Services at Virginia Tech .............................................. 5

Chapter 3: Related Work .................................................................................................. 6
  3.1 Emergency Notification ............................................................................................ 6
    3.1.1 Commercial Mobile Alert System ...................................................................... 6
    3.1.2 Service Providers .............................................................................................. 7
    3.1.3 Third-Party Apps ............................................................................................. 8
  3.2 Emergency Guidance and Incident Response .......................................................... 10
    3.2.1 Red Cross Apps ............................................................................................... 11
    3.2.2 FEMA App ....................................................................................................... 12
    3.2.3 ReadyVA App .................................................................................................. 12
  3.3 Conclusions and Thoughts ........................................................................................ 12

Chapter 4: VTGemini Design and Implementation ......................................................... 14
  4.1 Resource Loading and Management ......................................................................... 15
  4.2 Configuration Files and Resources .......................................................................... 17
    4.2.1 Callbox Locations ............................................................................................ 18
    4.2.2 Desk Reference ............................................................................................... 18
    4.2.3 Guide Configuration ........................................................................................ 19
    4.2.4 Welcome Screen .............................................................................................. 19
    4.2.5 Header Images and Guide Icons ..................................................................... 20
    4.2.6 Fetching .......................................................................................................... 20
  4.3 Custom Color Scheme and Layout .......................................................................... 20
  4.4 Collection View ......................................................................................................... 21
  4.5 Guides ....................................................................................................................... 22
  4.6 Sliding View Controller ............................................................................................ 23
    4.6.1 Expanding Menu Items ................................................................................... 25
    4.6.2 Search .............................................................................................................. 25
  4.7 Callboxes ................................................................................................................... 27
  4.8 Building Information ................................................................................................. 28
  4.9 VT Alerts Integration ............................................................................................... 31
  4.10 Developer Menu ..................................................................................................... 37
Chapter 5: VTGemini Evaluation

5.1 Accuracy
5.2 Functionality
5.3 Maintainability
5.4 Interoperability
5.5 Usability
5.6 Dependability
5.7 Performance

Chapter 6: Concluding Remarks and Future Research

6.1 Concluding Remarks
6.2 Future Research

References
List of Figures

Figure 1: Classroom Poster generated by OEM [Virginia Tech, 2012b] ........................................... 4
Figure 2: Overview of the CMAS architecture [Sprint, 2013] ............................................................ 7
Figure 3: View of the pull to refresh mechanism ................................................................. 16
Figure 4: An example guide .................................................................................................. 18
Figure 5: Initial test of a collection view ................................................................................ 21
Figure 6: Welcome Screen for VTGemini with the new layout and color scheme .................. 21
Figure 7: Initial guide list ........................................................................................................ 22
Figure 8: Quick Reference Collection View .......................................................................... 22
Figure 9: Initial test of the sliding view controller ............................................................... 24
Figure 10: Sliding View Controller showing the menu with expanded items ................. 24
Figure 11: Active search for the term "win" ......................................................................... 26
Figure 12: Initial callbox map design .................................................................................... 28
Figure 13: Final version of the callbox locations ................................................................. 28
Figure 14: List of all Virginia Tech buildings ......................................................................... 29
Figure 15: Building detail of a selected item ........................................................................ 29
Figure 16: Directions to selected building ............................................................................ 30
Figure 17: Final version of the building information popup .................................................. 30
Figure 18: Table view of active alerts .................................................................................. 32
Figure 19: Step-by-step guidance view .................................................................................. 33
Figure 20: Map view of an active alert ................................................................................ 33
Figure 21: Architecture for the development version of the VT Alerts system .................. 34
Figure 22: VT Alerts test administration portal ...................................................................... 35
Figure 23: Example Active Alert ............................................................................................ 36
Figure 24: Registering a device for VT Alerts ...................................................................... 36
Figure 25: VT Alerts Architecture and Workflow [Harris, 2011] .......................................... 37
Figure 26: Developer Menu for VTGemini ......................................................................... 38
Chapter 1: Introduction

1.1 Problem Definition

The nearly ubiquitous use of handheld, internet-connected devices in the United States presents an interesting opportunity for emergency and incident managers, and first responders. While there has been an increase in “big data” analysis through social media streams to analyze and understand the nature of incidents and tragedies (e.g., the Crisis, Tragedy, and Recovery Network (CTRnet) [Crisis, Tragedy, and Recovery Network, 2013]), there has been little research into using mobile devices to help guide and instruct citizens within the immediate area of a threat. There are some emergency managers who utilize social media to alert and instruct citizens about an ongoing incident; however, social media is subject to a lot of noise for individual users and there is no direct way to discern the difference between a tweet from a celebrity and one from a local crisis management office.

Furthermore, alerts and notifications that are issued from entities like the National Weather Service (NWS) do provide high-level, commonly-accepted instructions for citizens [National Oceanic and Atmospheric Administration, 2013b]. These alerts and notifications tend to apply to a county level and are, therefore, not granular enough for citizens within the immediate vicinity of an on-going incident. Likewise, individual corporations and entities within a county may require specific responses to incidents for its employees or dependents. While many of these entities may have detailed emergency action plans, these plans become useless in an emergency where a citizen may be required to act quickly by following a set of steps and list points.

An important part of emergency response management is the ability to minimize collateral and secondary incidents by providing guidance to members of the general public. During an eminent threat, guidance must be simple, clear, and concise to achieve maximum effectiveness. Additionally, educating the public on emergency preparedness through crime prevention initiatives prior to a crisis can help mitigate the scope of collateral and secondary incidents.

To address these problems, we have designed and implemented a mobile solution that focuses on users within an immediate area-of-interest for an incident. The software application was designed to focus on the specific needs and interests of a community, Virginia Tech, and to provide direct access to guides and information developed by the Office of Emergency Management (OEM) at Virginia Tech [Virginia Tech, 2013d]. Additionally, the application utilizes simplified step-by-step guides of the more verbose documentation on the proper response to an emergency by a citizen. Along with providing guidance to a citizen, the application supplies another method for notifying citizens of on-going VT Alerts through the Virginia Tech Emergency Notification System (ENS) [Virginia Tech, 2013b]. Finally, the application is not meant to replace existing OEM protocols but, only, to augment and enhance these protocols to improve their efficacy during an emergency situation [Virginia Tech, 2013e].

1.2 Objectives of the Software Application

The goal of this research is to develop a mobile software application (app) that addresses the problems of guidance and notification for specific threats and incidents within the Virginia Tech
community. The granularity of the domain is an important aspect of this app since it is designed to work specifically and exclusively with the accepted OEM practices of Virginia Tech. While the narrow domain of the app is not unique among mobile apps, it is unique among the class of apps geared towards emergency management. Specifically, the guidance protocols are unique and applicable only to the Virginia Tech community and it is not recommended that these protocols be enacted at other universities, business, and other entities. Finally, the notification system is maintained by Communications Network Services (CNS) at Virginia Tech and is only activated for incidents involving the Virginia Tech community [Virginia Tech, 2013a]. Therefore, users and citizens with no affiliation (geographically, personally, or otherwise) will find little utility in the app.

Additionally, this thesis catalogs and details the process in which we developed the app, with a focus on the design decisions we made in each iteration of development. In particular, we discuss several unofficial common user interface components and integrated JavaScript to assist in content generation and management. The design decisions and processes we chose are meant to inform and guide other mobile app developers in their own development life cycles with respect to certain requirements like network congestion mitigation and accessibility to content during a crisis.

1.3 Importance of the App

The stated problem is critical to any organization, especially campuses, which are responsible for a large population during an emergency. The ability to provide instruction and information to a group of people quickly and effectively will aid first responders and prevent secondary incidents. Additionally, because the number of smartphone users has increased with the proliferation of iPhones and Android devices, there is an opportunity to reach citizens in new and innovative ways during an incident. By creating an app that provides emergency response guidance, users will be able to gain access to important information regarding an active threat. The app can take advantage of the features offered in iOS devices like the Global Positioning System (GPS), a familiar user interface, network connectivity, and data management to provide context-sensitive instructions for the user.

We designed and implemented the iOS app with full cooperation and investment by the Virginia OEM and CNS at Virginia Tech. The goal of the app is to provide meaningful, helpful, and supportive information to citizens during an incident to minimize additional harm or damage, augment currently accepted protocols, and assist first responders through educating citizens on scene. The impact of the app is that of a more educated public that can potentially mitigate additional losses and save lives.

1.4 Thesis Overview

In Chapter 2, this thesis provides some background to the origin of the app and specific entities within Virginia Tech. Chapter 3 describes our analysis of the current landscape of emergency management apps. In Chapter 4, we discuss a detailed breakdown of the evolution and description of the app. Chapter 5 provides a self-evaluation of the app with respect to quality indicators. Finally, in Chapter 6, we state some conclusions and future work.
Chapter 2: Background

In this chapter, we provide background information on various departments at Virginia Tech that assisted us in creating VTGemini. We describe the work domain and brief description of the entity as well as their contribution to the app.

2.1 Virginia Tech Police Department

The Virginia Tech Police Department (VTPD) is responsible for students, faculty, and staff within the borders of the Virginia Tech campus [Virginia Tech, 2013c]. This includes law enforcement and crime prevention activities throughout the community. Many of their programs are designed to educate community members in self-defense through Rape Aggression Defense (RAD) classes and the Student Policy Academy. Additionally, they maintain lists of committed crimes and ongoing crime alerts [Virginia Tech, 2013f]. Their website and community outreach informs citizens on safety and emergency training while providing access to Safe Ride, a nighttime safety escort, and fingerprinting.

The VTPD contains a plethora of safety and emergency information that is useful during an emergency situation. However, disseminating that information in a timely manner is difficult due to network congestion and a lack of an appropriate medium for delivery. The idea of a mobile app assisting citizens during VT Alerts began to form. An officer of the VTPD generated a high-level design for a step-by-step guidance app as a “what to do” during an incident. The initial idea was to notify citizens through e-mail or text message with the guide to inform them how to properly respond to the ongoing alert. This idea was quickly eliminated in favor of a native app due to the uncertainty surrounding a consistently available network connection. During an alert, the network (both cellular and internet) tends to become highly congested. Moreover, a web-based mobile app was disregarded because of network concerns.

2.2 Office of Emergency Management at Virginia Tech

The Office of Emergency Management (OEM) at Virginia Tech is responsible for preparing, managing, and recovering from incidents and threats to the community. Along with internal protocols to ensure the continuity of operations at Virginia Tech, OEM provides many publicly available datasets for what to do in case of an emergency. Besides the online resources, every classroom includes a “Classroom Poster” (Figure 1) that provides a simple guide to specific alerts and other information [Virginia Tech, 2012b]. Additional materials include the Desk Reference which provides information for all possible alerts on campus and the Blue Light (callbox or emergency phone) map [Virginia Tech, 2012a; Virginia Tech, 2012c]. OEM also provides training programs and certification programs for coordinators and citizens alike.

OEM is one of the stakeholders and users of ENS. As such, they are responsible for the content, duration, and timing of each alert. Additionally, OEM manages the content of each of the alert guides. There are five major categories of alerts that can occur on campus: Hazardous Materials, Health, Facilities, Physical Threat, and Weather and Natural Disaster related emergencies [Virginia Tech, 2013g]. Each of these categories is further broken down into subcategories. The
guides provide detailed information about the incident, how to prepare for the incident, what to do during the incident, and how to recover from the incident.

Figure 1: Classroom Poster generated by OEM [Virginia Tech, 2012b].

While the online guides located on the OEM website are quite extensive and detailed, not all of the information is required during an active alert. During an ongoing incident, it is prudent to present citizens with concise yet informative information about the incident. Therefore, OEM provides a Desk Reference that is given out to citizens that contains information that can be quickly accessed and relayed during an emergency.
2.3 Communications Network Services at Virginia Tech

Communications Network Services (CNS) is the telecommunications service provider for Virginia Tech. They provide the “voice, data, and video services to … the University community” [Virginia Tech, 2013a]. Most importantly, CNS is responsible for maintaining and managing network infrastructure during a VT Alert by providing a secure, reliable channel to issue alerts to the community. All critical infrastructures must be maintained and operational during an alert to meet specific mandates and guidelines.

CNS has previously worked on and researched the integration of VT Alerts and iOS Push Notifications (APN) to provide another channel of disseminating VT Alerts [Apple, Inc., 2013a]. Currently, CNS has created a secure RESTful endpoint that will allow devices to register and, subsequently, receive notifications from VT Alerts [Fielding, 2000]. The goal of the endpoint is to create an API that any vendor with a notification architecture can subscribe to and receive alerts. Although we are using iOS Push Notifications, their system will be scaled to include Android Push Notifications or any other app that registers via the API.
Chapter 3: Related Work

This chapter provides an overview and analysis of the current mobile app landscape in terms of emergency notifications and guidance, and incident response. The analysis will help us understand the relationship between VTGemini and other emergency management apps.

3.1 Emergency Notification

There are two different types of emergency notifications that we address in this section. The first type of notification is based on a user’s service provider and the federal rule for the Commercial Mobile Alert Service (CMAS) [Federal Communications Commission, 2013a]. Most of the major service providers in the United States have adopted these standards, as have most handset manufacturers. The second type of notification is less restrictive and more informal. These notifications include local (county or state) level notifications about specific weather-related emergencies or other naturally occurring incidents. These types of notification can also include specific announcements within a certain county.

3.1.1 Commercial Mobile Alert System

CMAS was born out of the Warning, Alert, and Response Network Act, Title VI passed in 2006 [Federal Communications Commission, 2006]. The purpose of CMAS is to provide a new network layer to deliver emergency notifications to consumers. The network traffic on this layer is required to have at least equal priority to active data and voice transmissions to a device. Internally, Alert Messages sent via CMAS are limited to ninety alpha-numeric characters and must include the type of alert and severity. The FCC has, currently, defined three types of alerts that are allowed to be sent through the system (in order of importance): Presidential, Imminent Threat, and Amber Alerts [Federal Communications Commission, 2013b]. As an added capability, CMAS allows for geographically targeted messages in a specific geofence [Federal Communications Commission, 2013b]. This allows providers to be flexible in the target area and less restricted by jurisdictional boundaries.

While actual numbers and analysis of successful alert delivery are not published, the provision that alerts cannot preempt active voice calls (or any functionality on a device) and the mobile nature of cell phones it is possible that some CMAS alerts are never received by their intended target. That being said, there are several telecommunication companies who have dedicated resources to creating, maintaining, and administering an emergency wireless network that operates independently of commercial systems. One such company is TeleCommunication Systems, Inc. that provides extensive CMAS solutions and E911 services to the public [TeleCommunications Systems, Inc., 2013]. An overview of the CMAS architecture is shown in Figure 2.

Many of the most popular smartphone manufactures and mobile operating system designers include CMAS capability directly into new handsets; this includes all versions of Apple’s iOS devices running iOS 6 or higher. As part of its provisioning, CMAS is a voluntary service for both consumers and providers. A consumer can opt out of the service by simply turning off notifications provided that their service provider is CMAS-compatible. As stated above, most of
the major service providers in the United States support and promote CMAS compliance; however, many smaller service providers have opted out based on implementation costs, lack of consumer interest, geographic restrictions, or other reasons.

3.1.2 Service Providers

Prior to the WARN act ratified by congress in 2006, many service providers (especially service providers with a large market share) had implemented a Wireless Emergency Alerts system. As of 2012, Verizon, AT&T, and Sprint have upgraded their networks to comply with CMAS and they have rebranded their legacy systems as such. Since CMAS requires a hardware and software upgrade to existing devices, many older devices are not CMAS-compatible while newer devices (after 2007) include the necessary hardware and only require a software upgrade to receive CMAS alerts [Federal Communications Commission, 2013a]. Although the FCC ruling for CMAS outlines specific characteristics and requirements for service providers on how to implement and manage CMAS alerts, each provider implements the system in a different way [Federal Communications Commission, 2013a].

Verizon’s CMAS implementation only covers portions of its coverage area and there may be areas that are covered by Verizon but are not CMAS-capable [Verizon Wireless, 2013]. As a matter of course, a mobile device that is outside of Verizon’s service area cannot receive CMAS alerts. As per the ruling, their implementation allows for geographically targeted alerts which are
issued for each device in the target area. This means that if a user is travelling to another geographic area and their home area receives an alert the user will not receive that alert. AT&T and Sprint have very similar implementations as Verizon, if not identical, as per the requirements of CMAS [AT&T, 2013; Sprint, 2013]. An interesting implementation point for Verizon is that while they comply with the geographic targeting of CMAS messages, they also have the capability to issue alerts on specific radio towers [Verizon Wireless, 2013]. This effectively narrows the target area considerably and provides a high granularity of control of emergency notifications.

3.1.3 Third-Party Apps

There are a number of third-party apps that are non-CMAS compliant and are not required to be such. Many of these apps implement their own notification structures based on their own internal needs and rely directly upon a notification service. In this review of apps, we focus primarily on iOS apps as this provides a direct comparison to the VTGemini iOS app. These apps, like VTGemini, are subject to the reliability and stability of Apple's Push Notification services and infrastructure.

There are several types of emergency notification apps that can be found on the App Store. Some of them are for specific types of emergencies like hurricane, earthquake, or tsunami advisories and warnings. Since our solution includes emergencies of all different types for the Virginia Tech community, we primarily focus on apps with similar functionality and scope in terms of emergency notifications.

3.1.3.1 Wunderground

Wundergroud is an open-source aggregator of weather data and alerts from individual weather stations, airports, and Meteorological Assimilation Data Ingest System stations [Weather Underground, Inc., 2013a]. As part of an iOS app, Wunderground displays a current locale with information about the currently observed conditions and forecasts [Weather Underground, Inc., 2013b]. Additionally and most importantly, it advises the user about current alerts with a small notification image. The app does not actively notify the user of ongoing alerts when the app is launched or when the user switches locales. The user must access and view the alerts by clicking on the alert image. However, there is no indication to how many active alerts are ongoing or even the severity of those alerts.

Once the user has clicked on the alert image, the app displays a collection view of all active alerts in a nice overlay. Unfortunately, the content of these alerts are directly from the NWS albeit with some formatting to ensure some level of readability. The NWS alerts are notoriously written and published in all caps which make it difficult to read quickly. Additionally, the formatting includes a profuse usage of the ellipsis to communicate information. In its earlier years, NWS only issued current observations and short forecasts. Currently, NWS issues statements that include instructions and advice based on the type and content of the alert.

As an important note, if the app is running in the background or not running at all, these alerts are never seen by the user. In order to receive Push Notifications from the app for a particular
area, there is an extra step required: the user must go into the settings of the app and navigate to “Severe Weather Notifications”. The user must manually enter a location they wish to receive alerts in the form of Push Notifications from. There are several pros and cons to this approach, but overall we believe this to be a poor design choice for several reasons. The first is the unnecessary step of adding a location when it would be easier to add a “Subscribe to Alerts” button to the main screen. Secondly, it does not address the issue of a user moving to a new location. The current system requires the user to enter a new location every time they travelled. Congruently, the user continues to receive alerts from previously added locations which would diminish the usefulness of the app if the user is constantly bombarded with weather alerts.

Finally, the decision to use the alert text directly from the NWS can be a deterrent since the readability and intuitiveness of these text alerts is questionable. A concise and descriptive alert message would be far more effective to convey the same information. Overall, the app excels at its primary purpose, displaying weather information, but falls short on effectively providing users the ability to receive emergency alerts.

3.1.3.2 National Oceanic and Atmospheric Administration

While National Oceanic and Atmospheric Administration (NOAA) and NWS have not developed a native app, there is a plethora of apps that access and display data originating from these organizations [National Oceanic and Atmospheric Administration, 2013a; National Oceanic and Atmospheric Administration, 2013b]. NOAA recommends two iOS-based apps: Weather HD and Storm Shield [The E.W. Scripps Company, 2013; Vimov, 2013]. Weather HD is a free app that provides a feature rich and immersive environment with animations and 3D views of currently observed conditions at the user’s current location or other “favorited” locations. While the main focus of Weather HD is weather observations, it also includes the ability to send Push Notifications for severe weather alerts. Storm Shield is pay-for-use app that works directly with NOAA and NWS data and radio with a focus on weather warnings and watches for user selected areas.

As stated above, Weather HD is more of a currently-observed-conditions app for the user’s location and saved locations then an alerting mechanism. It includes very well designed animations depicting the current conditions, and a process-intensive 3D model of the earth showing the user’s saved locations and the current cloud conditions around the globe. Similar to the Wunderground app, it displays an alert at the top of the screen that originated from NWS. The placement of the alert banner is a poor design choice for several reasons. First, the new iOS notification center interferes with and hides the alert message when a notification is received. Secondly, the design of the alert message intuitively indicates that the user should “pull down” to view the alert. Unfortunately, this gesture is the exact same gesture required for the iOS notification “window shade” functionality and, thus, the user’s only option is to tap on the alert to view the details. The Weather HD app does a much better job of formatting the NWS alert message then Wunderground. The app properly capitalizes and removes unnecessary ellipses from the alert message. As for Push Notifications, the free version of Weather HD does not include this functionality. Similar to other apps, the user selects which locales they wish to receive notifications.
Storm Shield’s mission is to send emergency alerts to users based on the storm instead of the county [The E.W. Scripps Company, 2013]. Currently, most alert system issue warnings and watches on the county level with seemingly no regard to the movement and location of the storm in question. Storm Shield differs itself by analyzing the storm and issues alerts to users only when their location (or saved locations) fall within a polygon based on the storms meteorological characteristics. The app allows the user to receive or ignore almost every known NWS alert type from Flood to Fog to Air Stagnation, and enter up to five saved locales for a total of six possible locations including the user’s current location. Additionally, it is difficult to see the value in the app because the interface is simple, elegant, and informative. The added cost is the server-side analytics and weather tracking that is done to determine the approximate storm polygons and perform point-in-polygon analysis for all known users of the app.

### 3.1.3.3 CodeRED

Unlike the previous apps which are primarily weather-related with integrated Push Notifications, CodeRED is focused on alerts [Emergency Communications Network, LLC., 2013]. These alerts can include community-related, weather-related, or marine alerts. The free version does not allow you to receive alerts other than the community-related alerts. The app requires the user to acknowledge that it will be using Push Notification, but we were unable to find any documentation about the actual usage of this feature. The app includes settings for setting the alerts details, such as the sound to be played, as well as requirements for when to receive alerts. The overall design and look-and-feel of the app is amateur, at best, with a reliance on modal views instead of navigation controllers. The app fails to inform or elicit any emotional response from the user when it is being used which ultimately decreases its usefulness as a notification app.

### 3.1.3.4 Arlington County Alerts

The final app that we reviewed was the Arlington County Alerts iOS app [Roam Secure, Inc., 2013]. This app is different from the previous apps because it is designed specifically for a particular county, in this case Arlington, Virginia. As an important note, this app was not created or endorsed by Arlington County as it was developed by a third-party vendor [Roam Secure, Inc., 2013, p. FAQ]. Likewise, the data feeds and streams are of incidents that occur in Arlington but are not generated by Arlington County officials. Although this app is very simple, its use case and niche are compelling. It eliminates some of the complexity of a generic app for any jurisdiction and focuses solely on the needs and desires of a specific county. The app includes NWS warnings and watches for Arlington County, but it also contains Traffic Alerts with respect to accidents, congestion, and road work. Additionally, the app includes any county mandates or restrictions such as water or power based on current infrastructure conditions.

### 3.2 Emergency Guidance and Incident Response

Now that we have discussed the possible apps and mechanisms for receiving emergency alerts and notifications, we can review and analyze the current landscape for apps that provide information about properly responding to ongoing incidents and emergencies. Similar to the notification apps we discussed above, we attempt to analyze apps based on their granularity. For
example, the Red Cross and the Federal Emergency Management Agency (FEMA) apps are geared towards a national audience while the ReadyVA app is aimed for residents of Virginia [The American Red Cross, 2013a; USA.gov, 2013; Virgina Department of Emergency Management, 2012].

### 3.2.1 Red Cross Apps

The Red Cross has created several iOS apps that aid users in preparing and reacting to ongoing or upcoming emergency threats like hurricanes and tornadoes [The American Red Cross, 2013a]. Each type of incident is contained within its own native app. Each of the apps is identical in design with only differences in the content based on the type of emergencies. The Red Cross contracted the firm 3 Sided Cube from the United Kingdom to create each of these apps [3 Sided Cube, 2013]. The apps are very well designed and organized, and include guides, games, maps, and other information for users.

The Red Cross has four natural disaster apps: hurricane, tornado, wildfire, and earthquakes. The fifth app is First Aid. The design of each app is identical, so we will discuss the functionality of the apps from a high-level to reduce redundancies. The app includes guides for before, during, and after an incident which is similar to the VTGemini app for incidents at Virginia Tech. Each of these guides presents the user with a list of things to know and understand. Additionally, the guides include a checklist of items to complete before, during, or after an incident. For example, in the case of an impending hurricane, the citizen may want to fill up their car or board up windows and doors. Each of the guides also contains related links to additional guides and checklists. This app contains very deep integration with content management and relational connections with the content. This is evident throughout the app and especially within the guides’ dynamic and interactive views.

The apps allow the user to create and manage an emergency plan, complete with meeting and evacuation points, pet friendly hotels and emergency contacts. As part of teaching the user about a specific emergency, the apps include an instructional workshop in the form of a game where users can unlock achievements and badges as they progress through different steps and acquire new knowledge about an incident. The game is quite useful when learning and provides helpful answers to questions a user might have missed. The long-term benefit of including the game is questionable, but it does successfully keep users interested and interacting with the app.

Although we do not cover these apps in the previous section, they do include the ability to receive Push Notifications from several locations. In general, the first location will be wherever you first installed and ran the app. However, it is possible to enable “Monitoring Location” which will continually update and issue alerts based on the device’s current location.

The First Aid app is slightly different than the other app because it does not actively notify the user of any alerts and provides a far more interactive learning environment [The American Red Cross, 2013b]. The app is connected to the Apple Game Center to gamify learning first aid for specific injuries and emergencies. The app provides two versions of an injury: one that is a detailed reference guide, and the other that is a quick guide for treating a wound or injury.
While, individually, these apps are extremely rich in content, fun to use, and well designed, they lack full integration with each other. It is possible to open other Red Cross apps from within the current app via the integrated “Collect Them All” view. There is little to no data sharing amongst the apps, specifically when it comes to creating an emergency response plan. Additionally and similarly to other apps, the guides are generic and may not be completely applicable to specific locales.

3.2.2 FEMA App

The FEMA iOS app is simply a port of the mobile web site and stored locally on the device [USA.gov, 2013]. All of the views are displayed in a web view and content is displayed via HTML. The app contains many of the FEMA accepted definitions and guides with respect to emergencies and disasters. The guides are displayed in long lists of what to do before, during, and after an incident. The interaction and design is very simplistic and unintuitive. There is not much to analyze in this app because of the lack of interactive native views, but it is useful to be aware of such an app.

3.2.3 ReadyVA App

The ReadyVA iOS was developed by the Virginia Department of Emergency Management to provide information about possible threats and disasters in Virginia [Virgina Department of Emergency Management, 2012]. Many of the guides are generic, however, and can be applied to any state. These guides have similar content to our app as well as the Red Cross and FEMA apps. Similar to VTGemini, these guides are geared and tailored to the unique needs of the Commonwealth of Virginia. Similar to the Red Cross apps, ReadyVA allows citizens to generate emergency preparedness kits which consist of a checklist of items and tasks that the citizen wishes to complete during certain incidents.

Additionally, the ReadyVA app aggregates ongoing alerts for all counties within Virginia. It does not, however, include the ability to send Push Notifications to the user. There are actually three different views for alerts: the local view which displays alerts based on the user’s current location, the “Alerts” view which is a poorly designed table view for weather alerts and statements, and a third view which displays alerts from a user selected county in Virginia.

3.3 Conclusions and Thoughts

After reviewing and analyzing the apps, we conclude that the VTGemini app fits between the ReadyVA and Red Cross app in terms of design. VTGemini incorporates the ability to display local content and remote content in the same view. VTGemini is similar to the ReadyVA app in that our target audience is well-defined and, relatively small. VTGemini is designed specifically for members of the Virginia Tech community since it incorporates processes and policies that directly relate to the unique needs of the university. We will discuss our design decisions in the next chapter in further detail.

Additionally, emergency and disaster notifications in VTGemini are deeply integrated with the VT Alerts system and, therefore, incorporate Push Notifications similar to the Red Cross apps.
The Push Notifications that VTGemini receives are fine-grain messages directed at specific campuses within Virginia Tech, such as the Blacksburg main campus or the National Capital Region campus.

Finally, the guides that VTGemini provides to the user follow similar protocols from FEMA based on their guidelines, but are also modified to match the needs and requirements of the Virginia Tech community. VTGemini presents each of these guides as an overview, before, during, and after set of views.
Chapter 4: VTGemini Design and Implementation

We have designed and implemented the VTGemini app using Apple’s mobile operating system: iOS. We chose to create our app with iOS instead of other mobile operating systems, such as Windows Phone or Android, because the quality of the development environment provided by Apple is far superior to that offered by Microsoft or Google. The development environment and the implementation of certain technologies allowed us to rapidly develop high quality apps. Additionally, the device landscape is far simpler to navigate than the competitors, particularly Android devices, where we only need to develop and deploy to iPhone, iPod Touch and iPad mobile devices.

Our main focus was to create an app that addresses the primary needs of the stakeholders while providing an innovative and unique user experience that effectively communicates the emergency preparedness guidelines that have been created by the OEM at Virginia Tech. Therefore, we focused our efforts on developing a universal high-definition app for all versions of the iPhone, iPod Touch, and iPad mobile devices.

With our target devices in mind, we began to decouple the user interface with the knowledge base created by the OEM. The mobile app is agnostic to any changes to the knowledge base such that modifications to either the mobile app or the knowledge base can be done independently. In order to accomplish this, we needed to create a normalized property-list schema that represented the information found in the Emergency Information Desk Reference for Blacksburg as well as the emergency response guides from the OEM website to avoid ambiguity and simplify maintainability.

The normalized property-list files acted as the knowledge base for the mobile app. Each document contains data with regard to a particular incident and references to related topics or questionnaires for how a citizen should react to an incident. The documents were maintained on a server and accessed by a web-based administrative portal that allows authorized personnel to create, modify, or remove documents from a repository.

After the initial design and implementation of VTGemini, we decided that a new interface design with useful icons and layouts were necessary to create a positive user experience. Our software engineering design approach followed incremental development, iterative refinement, and progressive elaboration. At this time, we began working closely with OEM who had employed a designer to help create and maintain the images and icons used in the Desk Reference. The sharing of these resources was necessary to create a common user interface experience regardless of where the user was accessing the information. The user could visit the OEM website and see the same sets of icons as what is used within VTGemini. Additionally, the user could open the Desk Reference and see the same content that is generated with the app.

As the incremental design phase came to an end, we began working on a new design and architecture for the entire system. There was a large amount of new images and icons that were to be used within the app, and now we had the content of the guides to include. We still needed to create a universal app for all of the devices; this is a requirement that was non-negotiable since we are trying to reach as many members of Virginia Tech community as possible.
We eliminated the need to generate a separate normalized knowledge base of the guides since we now had access to guides from OEM. We refocused our efforts by creating index-like property-list files that organized and linked content with each other appropriately. The property-list files also allowed us to create a dynamic interface where it was easier to modify the content by simply changing the property-list configuration.

Similar to the initial design, these configuration files were stored on a web-server and accessed on-demand. Later in this chapter, we discuss the configuration files in more depth. In addition to the configuration files being stored on the web-server, there was a time where all of the resources were stored remotely as well (we discuss the design choice later in this chapter). Furthermore, we discuss the commonly implemented user interface components that we used through our app later in this chapter.

### 4.1 Resource Loading and Management

As stated in our overview and design definition above, we wanted to incorporate commonly accepted design components and third-party libraries to assist with the display and management of content. A commonly used component in iOS design for table views is the idea of “pull to refresh”. It was first introduced by Loren Brichter, the developer behind Tweetie, in 2008 as a way to fetch new tweets for a user [Brichter, 2013]. The app has since been bought by Twitter and is now the current Official Twitter app. The gesture has since become ubiquitous in mobile apps that load content remotely. The intuitiveness of the gesture is the key to its success; once a user has performed the gesture on a table view, the user almost expects all table view to react accordingly.

Since the initial design of our app required the system to fetch configuration and property-list files remotely to be displayed in a table view, this gesture was a logical choice to implement. In Figure 3, we show the act of pulling the table view down to review the view for refreshing the content. Once the user has pulled the table view passed a certain threshold, the view controller is notified to reload and perform any operations necessary to grab new content. It is important to note that this gesture is not needed in environments where data is static; it is most useful in dynamic data sets that are primarily loaded asynchronously from a remote web-server. In the case of dynamic content that changes locally on the devices, the usefulness of the “pull to refresh” gesture is trivial since we can easily notify the view to reload its data if the underlying mutable data is modified.

Since its introduction, many developers have created various designs and code-behind implementations to achieve similar results. In the release of iOS 6, Apple introduced the new `UIRefreshControl` that allows user to fetch content on-demand [Apple, Inc, 2013b]. The gesture is very much the same as the original, but it does include a fancy tear-drop animation to indicate when the refresh will execute.

As we have discussed, the number of resources included with the system is very large. As such, our initial design of the app was to host all of the images, icons, configurations, and guide details on a web-server that would automatically download, update, and store resources that had been
changed or modified. In fact, we had created an entire architecture to support the ability to fetch content remotely.

The only configuration file and resource to be included in the app was a master resource list that maintained the version and file name of every resource used within the system. On the server-side, there was another resource list that maintained the current version and name of a resource. At startup, the app would compare these two files to determine what, if any, resource needed to be downloaded or modified. While this is a simplistic method of version control for resources within an app, our primary concern was space on the device and ensuring the size of the app did not exceed the 50 MB limit imposed by Apple [Apple, Inc., 2013c]. While the size of an app can most certainly exceed 50 MB, this limit forces users to have an active WiFi connection in order to download an app. Once the architecture and system were implemented, it was simply a matter of caching and downloading the resources as they were needed by the app. We believed that this would allow us to dynamically update and modify content remotely without having to submit an update to Apple’s App Store.

![Image of pull to refresh mechanism] Figure 3: View of the pull to refresh mechanism.

However, we eventually moved away from this architecture that supported remotely loaded content, with the exception of a few resources, for an entirely local implementation. The three main reasons for loading and managing resources locally are network concerns, OEM guide release and review schedule, and offline accessibility and performance. Each of these reasons
were realized and implemented at different points in the life cycle of the app. For example, offline accessibility has been a concern since the inception of the app.

One of the most prominent network concerns that we encountered was network congestion. Network congestion is a huge concern for this type of app since we are loading many individual icons and images. With several thousand potential users, the impact on servers at Virginia Tech is potentially crippling. This is even more prominent problem during an actual emergency situation because even more people will be accessing the app to download content. Additionally, whenever a VT Alert is issued and a legitimate emergency situation has arisen on campus, the servers instantly become overloaded by members of the community and the country-at-large as try to access Virginia Tech resources and services to gain more information about the ongoing threat or incident.

The second reason to eliminate remotely loaded content has to do with review and release schedule of OEM guides. We determined that the guides and other information are updated at most quarterly; although it is more likely that the guides are changed only twice a year. Since the update schedule is infrequent, the need for additional infrastructure to support frequent updates to resources is diminished. Therefore, there is a benefit to keeping a single copy of the guide locally on the device and update the app as needed.

The final reason to maintain the resources locally is offline accessibility and performance. One of the requirements for our app was to be able to access guides even if there is a complete network failure, which is common during a developing emergency situation. The only way for this to work properly is to store the content locally on the device. In our previous architecture, we loaded the resource as needed and stored them locally to increase performance. Since some citizens will only use the app during an emergency situation, this would mean they would be unable to access resources during an emergency situation and the network failed. Therefore, offline accessibility is crucial to the apps operation. Finally, by loading and accessing local resources we increase the overall performance of the app by avoiding lag and load times over the network.

As previously stated, all of our critical resources are stored and loaded locally on the device but there were other external resources that we did load remotely. The details of some of the resources will be discussed in subsequent sections such as the difference between local and remote guides. Furthermore, some of the resources for web-based API’s are loaded remotely like that of the Google Maps JavaScript API.

4.2 Configuration Files and Resources

There are several configuration files that are used by VTGemini to manage and display content within the system. While these have changed and evolved over time, their essence was first created in the initial design of the app. This section aims to describe each of the configuration files and their relationship within the app. These descriptions are meant to be informative but also contain technical details to assist future developers in maintaining the app.
4.2.1 Callbox Locations

The callbox locations configuration file is fairly straightforward: the key for each entry in the dictionary is the name (or descriptive name) of the location of a blue light emergency phone. Each entry is an array with two items: latitude and longitude. These locations were located manually by using the callbox map offered by OEM.

4.2.2 Desk Reference

The desk reference configuration file includes all known guides that are in the Desk Reference. This is the most important configuration file as it provides details about the offline guides and how to access these guides. Each entry contains the following five keys: Image, Highlighted, Name, Header, and Link. Image and Highlighted provides a reference to the name of the image resource included in the app that represents the given guide. Name is the descriptive name of the guide. Header is the image reference to image displayed at the top of the guide. Link is the name of the HTML file that contains the content we wish to display for the selected guide.

Figure 4: An example guide.
In Figure 4, the Preparedness guide is displayed. The header image is at the top and colored green. The body of the guide is an HTML file that shows the content where the headings are styled with the same color. The HTML files stored locally by the system are the Desk Reference content files. Each one is surrounded by a unique div that assists with the styling of each file based on the defined color scheme. This color scheme can be found in the `global.css` file.

### 4.2.3 Guide Configuration

The “Guides” configuration file is, by far, the most complex of files because it maintains the hierarchical structure of the menu and submenus. A menu that contains submenus contains a key called “Objects” which contains more menus. Each menu item must contain the following: Image, Highlighted, Level, and Name. If the submenu contains a link-out without any submenus, then it contains the Web-Link property instead of the Objects property. The app automatically assumes that a menu with the Objects property contains submenus, so even if a menu with the Objects property also contains a Web-Link property, then the Web-Link Property is ignored. A menu item that is a link-out may also contain the CSSDivName property which defines the color scheme for the current menu item.

A menu item may also contain a Level and Order properties. The Level property indicates the indentation level; the indentation level is a fifteen point indent from the left side. Therefore, a Level of zero means no indent while a Level of one is a fifteen point indent. The Order property is a secondary measure to ensure that submenus are correctly organized when they are displayed by a collection view or a table view.

### 4.2.4 Welcome Screen

The “Welcome Screen” configuration file is used to properly organize the main collection view. Each section is represented by the first-level key name, with rows defined within each section. The overall organization of this document is directly mirrored in the app (See Figure 6). Each entry in the dictionary is slightly different because of these actions. All action types include an Image, Highlighted, and Color property. The Color property is the hexadecimal value of the primary color of its image.

The first action type is a simple segue. An entry of this type needs to provide a Segue Name. The Segue Name refers to the name of a segue attached to the welcome view in the storyboard file. Image and Highlighted are similar to other configuration files. The second action type is an internal link. An entry of this type requires the Link property. This can either be to a property list file or an HTML file. If it is a property list file, then the app automatically generates a new collection view to display the contents of the file. The third action type is a link-out. An entry of this type requires the Web-Link property. This property, unlike Link, cannot be a property list and must be a valid URL.
4.2.5 Header Images and Guide Icons

The Header Images group includes all the images needed when displaying a particular guide that has a defined Header property. There are three versions of each header file: normal, @2x, and ~iPad [Apple, Inc., 2013d]. The last two versions are identical, but older non-retina version of the iPad load the lower resolution image (normal) instead of a high resolution version which it can display properly. The Guide Icons group includes all the images required by various guides and sub-guides. There are two versions of each guide icon: normal and @2x. The first version is used for non-retina iOS devices while @2x is used for retina devices.

4.2.6 Fetching

The Fetching group includes auxiliary files that help load files and data from the network. The maps.html file is used by the building information view that we discussed section 4.8. It accepts specific parameters that allow us to generate the callout feature with information from VTGemini. It also is responsible for searching the DOM of a given webpage to determine the metadata description of a given building. It also contains additional supporting files for jQuery and the required CSS styling.

4.3 Custom Color Scheme and Layout

In order to match the branding requirements for Virginia Tech, we implemented our custom design and color scheme for the entire app [Virginia Tech, 2013h]. The primary colors are burnt orange and Chicago maroon which are the official colors of Virginia Tech. We also included a few other colors to help provide context to certain items within the app. For example, Figure 6 shows the home screen where each section as a defined color scheme. The icons and the font color are defined by a specific color as defined by OEM. Additionally, the colors used in the app match the colors in the Desk Reference to provide visual cues that we are indeed accessing the same content (See Figure 4 and Figure 6).

Each screen contains a custom navigation bar and tool bar. In our implementation, we overrode the drawing of each of these views in order for us to insert our custom drawing. For the navigation bar, we are using the Core Graphics package from Apple to draw a Chicago maroon rectangle with a burnt orange stroke at the bottom [Apple, Inc., 2013e]. For the tool bar, we perform a similar operation except the stroke is drawn at the top. Additionally, we used the Core Graphics current context to insert the Virginia Tech logo and the envelope icon. The envelope icon acts as a button since we added a tap gesture to the image view containing the icon to launch a mail composer view from within VTGemini.

As a continuation of the color scheme, every screen has a very soft yellow hue to help minimize eye strain generally associated with bright white backgrounds. Additionally, the table and collection views implement custom section headers of Chicago maroon with white text.
4.4 Collection View

As part of the new design, a custom menu structure was implemented such that we eliminated the use of a tabbed app. A tabbed app requires a tab bar at the bottom of every screen eliminates useful screen real estate. With that in mind, we decided to use a collection view to display different menu items on the home screen in a logical manner. Figure 5 shows our initial test of the collection view for the “Welcome” screen. The content of the view is loaded directly from a property-list file that can be modified to create a new menu structure without changing the code of the app. Figure 6 shows the redesign of the collection view with the custom color scheme and new layout for the Welcome screen.

![Figure 5: Initial test of a collection view.](image)

![Figure 6: Welcome Screen for VTGemini with the new layout and color scheme.](image)

When a user selects a menu item, the system reads the model associated with the item and segue based on what is defined in the model. The content of the next view after a segue is determined by the link within the model. For example, if a user taps on the “Guides” menu item the system navigates to another collection view and displays all known guide objects. These guide objects are loaded directly from another configuration file that defines each guide model. This is a
simplistic approach to generating content and menu items, but it created a more user friendly interface and provided context to each of the items.

4.5 Guides

If there are no active alerts, then the user can access guides and peruse the app to view each of the guides and knowledge base articles. In Figure 7, the app is showing a list of all known guides. The detail disclosure button shows what guides are available and will be downloaded from the web-server. When the detail disclosure button is selected, the system provides the same user experience as if the user had selected the guide from an active alert. This allows the user to gain experience with each of the guides prior to any active alerts.

![Figure 7: Initial guide list.](image)

![Figure 8: Quick Reference Collection View.](image)

In previous sections, we have discussed that most resources are stored locally on the device so that they are available during network disruption. Likewise, we have mentioned that a few resources are loaded remotely. We perform a similar operation for some of the guides.
Specifically, the non-Desk Reference guides are fetched and parsed directly from the OEM website to provide users with a detailed and up-to-date version of the guides.

When the user taps on a guide that needs to be remotely loaded and parsed, the app utilizes jQuery to obtain the DOM object of the page being requested. It then searches for a specific div that contains the content of the given webpage. Fortunately, Virginia Tech uses a common CSS class for all of their hosted webpages which allows us to easily find the required content in the page. Once we determine and fetch the required HTML from the page, we insert a custom div class that we have defined to provide styling and other features required for the app.

As such, the app contains locally and remotely sourced content for the guides and to represent the Desk Reference (See Figure 8). In the event of an active alert, however, the app disables the ability to load remote guides. The reason for this deactivation is two-fold: minimize network congestion and direct users to the Desk Reference. We have previously discussed the possible increase in network congestion during a VT Alert, and, thus, we are trying to minimize the impact that users of the app may have on critical infrastructure. Secondly, during an alert, we attempt to direct users to access the Desk Reference guides, which are stored locally, so that the user has immediate access to clear and concise information.

4.6 Sliding View Controller

The Sliding View Controller (SVC) is another user interface component similar to the “pull to refresh” component. SVC allows the user to move the top view to reveal a second view below the first. There are many use cases for such functionality. In our case, we use SVC to display a table view that displays the current active alerts and a quick link to all known emergency guides. We modified Michael Enriquez’s ECSlidingViewController [Enriquez, 2013] and used it as our SVC in our app. His implementation works for both left and right pan gestures to reveal two different types of view under the left or right side of the main view. Additionally, his implementation scaled to other devices like the iPad and iPhone 5 and worked perfectly with their screen sizes. The basic implementation of SVC is shown in Figure 9.

Despite its simplistic nature and well-done implementation, we did make a few modifications to the code to make it work and respond appropriately to specific events. The first modification we made to the implementation was removing the requirement to use the static class UIImage+ImageWithUIView. The purpose of this class was to take a screenshot of the main view prior to panning the top view. The screenshot was then used as an overlay to the main view. The reason behind the purpose of this implementation remains elusive. By creating this overlay, SVC effectively eliminated the ability for the user to interact with the main view. For example, after the user panned the top view to reveal the bottom view, if the user attempted to click the “Menu” button to bring the top view back into focus there would be no animation. The button would fail to show its highlighted state because the user was simply clicking on a static image. The desired effect of bringing the top view back into focus was achieved by the implementation adding a tap gesture to the image view that contained the screenshot. We found the static image to be useless and a waste of processing time (although it was minimal). Additionally, we wanted the top view to still allow users to interact with it and to animate in response. By removing the static image the “Menu” button correctly shows its highlighted state when tapped, and the
collection view can be scrolled even opened state showed in Figure 9 where the top view is panned to the right to display the bottom view.

The second modification we made to the SVC was to allow the user to pan the top view to open and close it. The code and implementation details are included to allow the user to pan, but in the version used for our app, the connection was not made. Additionally, it was possible to over-pan the top view to reveal a blank canvas between the right-edge of the bottom view and the left-edge of top view. We modified the pan gesture animation to prevent the user from over-panning by fixing the width to the value specified by the user at initialization. Essentially, the user could not pan the top view further to the right then the width of the bottom view.

The final modification we needed to make to the SVC was how to properly display the search bar, keyboard, and search results when it was activated. When the user taps on the search bar, a keyboard is displayed and search result table view is shown. Since the search bar is on the bottom view, the width of the search results table view is limited to the width of the bottom. This causes a very strange effect to happen because the top view is still visible. Therefore, when the
user taps the search bar to begin a search session we have to dynamically hide the top view and resize the bottom view simultaneously to create a seamless transition from one state to another.

Additionally, this transition could not refresh and layout the subviews because it was done within an animation loop. When this happens, the content of the table view (specifically the indentation of menu items) responds by animating itself. In our case, the indentation was animated so it looked as though the menu items were sliding from left to right. To eliminate the reloading of subviews, we had to avoid changing the width during the animation which required changes to a few lines of code in the procedure responsible for anchoring the top view. The result is a fairly smooth transition from a search session to a browsing session.

The SVC is one of the most common features in current iOS apps. Facebook and Google both utilize it to allow for more screen real estate for other content and actions. For our app, it allows user quick and easy access to guides and active alerts.

4.6.1 Expanding Menu Items

As the complexity of the app increased and the depth of the guides were researched further, it became apparent that we needed to implement a menu structure that allowed users to drill into specific categories of guides. The added bonus of properly indenting the submenu items turned out to be a blessing and a curse. We discussed the reasons for why it was a curse in the previous section with respect to the sliding view controller. It was a blessing because it helped us to decompose our ever-expanding configuration of emergency guides.

There are several iOS apps in the market that utilize the expanding menu items idea. However, most of these seemed to be implemented to respond to tap gestures on the section headers of a table view. Our implementation was done on the table view cells themselves. We performed several design iterations to properly implement a method to represent and indicate that the menu item contained submenus as opposed to an item that would display content. The final verdict and decision is shown in Figure 10.

Figure 10 shows how we have expanded “Natural Emergency” and its submenu “Earthquake”. We decided that to properly indicate that a menu item contained submenu items was to utilize a downward facing chevron (or, in iOS terminology, a disclosure icon). We can see that other menu items like “Flooding” and “Hurricane” have a chevron that indicates there are more items within the menu. Similarly, the chevron animates and rotates 180° to indicate that the menu item has been expanded and can be collapsed if it is tapped again.

4.6.2 Search

In the previous section, we discussed the modification of the sliding view controller to allow for properly switching to a search session. Figure 11 shows an active search session with an example search term entered in the text box. We can see that the top view of the sliding view controller is invisible and the bottom view, which includes the search bar and results table view, now fills the entire screen. The search function is fairly simple and allows user to filter all known guides
based on a specific query. The query and results are filtered as case insensitive to provide relevant feedback to the original query regardless of how the user entered it.

Figure 11: Active search for the term "win".

Since the bottom view displays both the search results table view and the menu table view, the underlying view controller used by the bottom view is responsible for both table views. That means that it must respond to the display controller for search and our custom table view. In order to implement this properly, we had to realize that the table view delegate for our view controller responds to calls from both table views. Therefore, our implementation need to check what the current calling view was and verify whether it was the search results table view or the menu table view. Fortunately, the delegate provides a reference to the calling table view and we can simply match the reference to the search display controller to verify it is the correct table view.

While this implementation is not ideal, it allows us to minimize the footprint of this particular view by storing only a single reference to all known guides in the view controller and, subsequently, allowing the search delegates to access that reference directly instead of reloading.
the content from the configuration files. This increases performance and the responsiveness of
the search features which automatically generates results as soon as the user begins to type in the
search bar.

Furthermore, the results table required a way to resign programmatically after a user taps on a
specific emergency guide. The system responds by completely resigning the first responder (the
keyboard), and deactivating the search display controller. This allows the bottom view to return
to its default state prior to navigating to the selected item.

4.7 Callboxes

The callbox view is the only view to have survived to the final product. Figure 12 shows the
callbox view with every blue light emergency phone on campus. When a pin is selected, the
callout view shows the distance to the user’s current location and its commonly used name by
Virginia Tech. The view automatically zooms and animates the dropping of each pin. Although
this is a very simple view, it effectively shows each location with respect to the user’s location
(in this case, the current location is off the current view). As an important note, this particular
screenshot was taken when Apple still utilized Google Maps are the default mapping API.
Currently, the map defaults to Apple Maps although the same context still exists.

We have discussed how the callbox view is the one view that remained constant throughout the
lifetime of the project. However, we implemented the ability for the app to determine closest
callbox and magnify the map to an extent based on the user’s current location and the closest
callbox. In the previous version, the extent was based on the minimum bounding box that
included all of the callboxes and the user’s current location. Additionally, Figure 13 shows the
change in the default map from Google Maps to Apple Maps. It is clear that the new mapping
system in this screenshot lacks the detail from the previous image (Figure 12).

A final piece of functionality was added prior to finalizing the application is show in Figure 13
as the walking directions to the closest callbox. These directions are decoded directly from the
Google Maps Directions API and draw as a line overlay onto the Apple Maps MapKit in iOS.
Additionally, each of the callboxes is represented by a small icon instead of a simple pin marker.
4.8 Building Information

The building information view allows users to scroll through the entire list of Virginia Tech owned and sponsored buildings. This list was provided to use by University Relations at Virginia Tech (See Figure 14) [Virginia Tech, 2013i]. The list contains buildings from all campuses including Blacksburg and the National Capital Region. Our app does not include all buildings from the list because there are entries that lack a campus designation and GPS coordinates. We are only interested in well-known and established buildings on campus and being to geolocate them accordingly.

If the user taps on a specific building in the list, a web-map view displays the current location as a pin. The user can click on that pin to view a callout with more detailed information about the building (See Figure 15). This description of the building can come from two different sources depending on the availability. Each building in the list is maintained as a model in the app, each model has properties that hold information about the building itself. One of those properties is a direct link to the University Relations website that provides a detailed explanation about the
building. Our app uses a custom JavaScript function to extract data from the HTML of a given building. The JavaScript function either pulls a description from a metadata tag, or it scrapes the body of the HTML and pulls out the first paragraph it finds in the main div of the document. Either of these paragraphs and descriptions is sufficient to describe the building. Once we have the description, we then display the callout with the building’s name, abbreviation, type, description, and an image.

![Figure 14: List of all Virginia Tech buildings.](image1)

![Figure 15: Building detail of a selected item.](image2)

Additionally, the building detail view that shows the web-map for a specific building allows the user to fetch walking directions from their current location to the location of the building. The user simply taps on the “Walk” icon in the upper-right corner and the user is taken directly to the Apple Maps app.

As an important note, we can see in Figure 15 that we are not using the built-in Apple Maps SDK and have opted to use the Google Maps JavaScript API [Google, Inc., 2013]. We have decided to do this because the level of detail provided by Google Maps is still far deeper than the new Apple Maps software. It also allows us to dynamically create a callout based on the current
building. We stated above that we scrape the webpage for the building; this is done quite easily using jQuery instead of requiring a third-party library to search and modify the document model of a specific webpage [The jQuery Foundation, 2013].

With the introduction of the Google Maps Directions API for the callbox view in the previous sections, we decided to implement this functionality for the building information view. The new implementation no longer requires the Google Maps JavaScript library to display the building. Instead we utilize Apple Maps and draw the directions polygon directly onto the map itself as show in Figure 16. The new view for building information allows the user to fetch walking and driving directions by using the toggle buttons. Additionally, the app allows the user to perform a long press (e.g. press-and-hold) on the toggle buttons to obtain turn-by-turn navigation from the native Apple Maps application in iOS. Finally, the change in functionality also required a change to the detailed view of particular building. If a building contains additional information as provided by University Relations, the pin annotation will include a detail disclosure accessory on the callout (see Figure 16). When selected, the resulting view will be that seen in Figure 17, which displays a modal view with the name of the building, abbreviation, an image, and a brief
description of the building. It is important to note that the functionality of being able to view detailed information of a building is disabled when there are active alerts.

4.9 VT Alerts Integration

The initial design of the app was a tabbed app that displayed Active Alerts, Guides, Callboxes, and Education (See Figure 18). The Active Alerts table view included a custom table view cell that displayed the type of alert, date of alert, a description, a button to view the associated guide (in Figure 18, the bell icon with a checkmark), and a button to view the alert on a map (in Figure 18, the WiFi symbol). The Active Alerts table view was split into several sections depending on the priority and urgency of the alert: immediate, urgent, or all-clear. The initial tab to be viewed was the Active Alerts view so that the user was always informed of the current state of the community. The Active Alerts were loaded directly from a property-list file hosted on a web-server. This property-list needed to be modified and the app notified of the change when a new alert was issued. The initial plan was to allow the VT Emergency Notification System (ENS) to dynamically modify this file when a VT Alert was issued. Subsequently, ENS would issue a Push Notification to VTGemini which, upon receipt, would notify the app to download a new version of the Active Alerts property-list.

We ran into several issues with this design because of the reliance on a single property-list file. The single property-list file created a single point of failure for the app. If that file was corrupted or missing, the app failed to initialize and ceased to function. Several steps were taken to help mitigate the potential for failure such as allowing the app to operate even with the file missing. However, it was obvious that a different solution was required for managing remote data that changes often.

As stated above, the user is presented with the ability to perform two actions on any given alert: view the guides associated with the alert or view the alert on a map. The first action is shown in Figure 19 and shows the step-by-step guide approach that we initially developed. Each guide was stored with a reference to the alert type which allowed the app to access and load the correct guide when alert was received. The alert types are based directly on the accepted types defined in the ENS protocol. The guides were also stored on a web-server with each guide broken into a separate individual file and a single index property-list that contained the name of the guide and its associated alert types.

Once the app loaded the guide, it would display the first action or question defined in the guide. In Figure 19, this first step included a question (“Did you find a door that locks?”), a set of answers, and some additional information. The user can proceed down a particular workflow based on the answers to the prompted questions. From a technical point of view, each answer generated a new view controller which was successively pushed onto the navigation controller. While, in theory, the idea of step-by-step guidance through a particular guide is attractive since it displays only the most pertinent information at any given time, in practice this type of interaction is very slow and too simple for the user to realize any value from the app. Additionally, asking questions instead of issuing commands to the user tends to slow the reaction time of the user as well. The goal of the app is to provide clear, concise, and thorough information as quickly as
possible without fanfare or distractions. The step-by-step guidance, while thorough, was distracting and the limited responses to nontrivial questions tended to confuse and inhibit users.

![Figure 18: Table view of active alerts.](image)

When the user tapped on the WiFi symbol to display a map view of the active alert, the view in Figure 20 was displayed. While not visible in this screen shot, the user’s current location was also displayed on the map to provide context to the active alert. The map view, while simple, provided useful information about where the active alert was occurring. In most cases, an active alert included a well-known name for the location of the alert. However, the alert did not include exact coordinates that are needed to place the pin on the map view. Therefore, VTGemini performs analysis on the location details of the alert by reverse geocoding the location name and determined the correct GPS coordinates for placement. The app makes no guarantees that this is the exact location of an ongoing incident and, therefore, utilizes common landmarks and places as the name of the pin to indicate that the incident is within the vicinity.

In order to test Push Notifications for VTGemini, we set up a custom architecture for registering and notifying devices in VTGemini. This was created prior to the production level integration with CNS and used as a test of the system to verify that it was possible to incorporate VT Alerts into the app.
Figure 19 shows the high level architecture used as an experimental implementation of VT Alerts. The Gemini Portal was built entirely in Ruby on Rails and hosted via Phusion Passenger on an Amazon Web Service Elastic Cloud Red Hat Linux instance [Hansson, 2013; Phusion, 2013; Amazon Web Services, Inc., 2013]. We choose Rails as the framework because it allowed us to implement a web-based administration tool (See Figure 22) while providing “for free” a RESTful web-service that would allow us to easily activate and deactivate devices, and provide access to the active alerts.

In Figure 21, the Gemini Portal maintained a device and alert database. The device database was populated with the Push Notification token generated by the app when the system requested access to remote notifications. After the device was registered, the Gemini Portal was responded with a properly formatted HTTP response that included a Gemini ID that was then used to verify the identity of the device. The developer menu in Figure 26 shows where the ID would be displayed if the device was successfully registered.
The Gemini Portal allowed an administrator to generate an alert in the browser and send that alert when necessary. The portal includes the `apn_on_rails` gem that allows the system to issue Push Notification via Apple’s Push Notification (APN) server [PRX, 2013]. We are able to do this because of the provision profile created for VTGemini which allows for development Push Notifications to be sent to a device. The gem also allows for us to group devices into Push Groups and then send notifications via these groups to reduce potentially multiple HTTP connections to only a single connection. Fortunately, APN allows a user to open a single connection and push an alert to as many devices as needed.

Once each of these alerts is sent, a notification message is displayed on the device. Once the user acknowledges the alert, the app sends a request back to the Gemini Portal requesting all known active alerts so that it can properly display each of these inside the alert view (See Figure 23). The alert view displays the type of notification, the description obtained in the custom attributes of the alert, the date posted, and the associated guide image. Although not visible in Figure 23, there is a map view that displays a pin based on the location of each of the active alerts. Now that the device has displayed the alerts, the user can scroll the collection view between the different alerts. The alert view also allows the user to view the associated guide to the current alert. This guide is tied directly to the Desk Reference to ensure that the user is getting the most concise and meaningful information from the app.
After the initial test of the VT Alerts system, the development group at CNS began architecting and implementing a system that integrates directly with the current VT Alerts system. The registration and notification service developed not only services iOS devices, but also Android and other registered vendors. The group has developed an Apache web-service that accepts incoming requests and responds appropriately.

As part of the design, the web-service requires mutual authentication between a device and the endpoint. This mutual authentication is based on several signing certificates from both CNS and Apple. The first certificate is generated by the CNS Certificate Authority and allows the app to securely connect to the web-service. The second certificate is generated by the provisioned profile for the app and stored locally on the device. The third certificate is used by the Push Notification service; this certificate is registered and stored at CNS to allow the app to register itself and the notification service to send out Push Notifications to registered devices.
The CNS web-service is planned to include two different versions of the endpoints: a sandbox and production environments. Both of the architecture are identical except for the production system includes more redundant servers and load balancers. Additionally, the registration and notification services are completely disjoint to prevent a crash of the entire system if one of the services fails. These services, while providing similar functionality, have a vastly different architecture and implementation than our initial test for VT Alerts. We can see in Figure 25 the overall workflow and architecture as envisioned by CNS for the registration and notification service. Here is an outline of the numbered interactions:

1. **Download and Install** – the app is installed from the Apple App Store
2. **Register Device** – the device registers its remote notification token with the CNS registration endpoint
3. **Issue VT Alert** – emergency managers issue a new VT Alert to the alert system
4. **Alert Queue** – the alert is sent to the broker which sends requests to different systems such as SMS, e-mail, and Apple Push Notification
5. **Notify APN** – the CNS service send the generated push notification to the APN servers
6. **Notify Devices** – APN servers send notifications to each of the devices
7. **Send Receipt** – the app sends a read and received receipt back to the notification server to verify the user has acknowledged the alert

The library generated by CNS provides all of the abstractions and delegates required to fetch the proper certificates and set up the correct secure HTTP requests to the web-service endpoints. In order to integrate the library into our app, we created delegates in our controller classes to handle the callbacks from the library. For example, our registration service needs to be able to display the terms and conditions of the current web-service prior to continuing the registration process. Once the user has agreed to the terms and conditions, the library will contact CNS and register our device token and notify our controllers that the registration is complete (See Figure 24).

![Diagram of VT Alerts Architecture and Workflow](image)

**Figure 25: VT Alerts Architecture and Workflow [Harris, 2011].**

### 4.10 Developer Menu

In the initial release of the app, we have included a hidden developer menu that allows the user to participate in development activities. Currently, our developer menu is used to enable the receipt of VT Alerts. Therefore, receiving VT Alerts currently defaults to an off state unless the user accepts the terms and conditions of the test and what it means for a VT Alert. This allows us to avoid some of the possible legal ramifications associated with improperly distributing an emergency notification.
Figure 26 shows the Developer Menu when it is activated. It allows the user to toggle developer mode and displays whether the user is registered to receive VT Alerts. In this particular screenshot, there are a couple of other housekeeping items that allow us to track the usage and build of VTGemini. The “Gemini ID” heading was used during the initial development phases of the VT Alert integration and is set to expire and be removed prior to the release of the app into the App Store.

Figure 26: Developer Menu for VTGemini.
Chapter 5: VTGemini Evaluation

This chapter provides a self-evaluation of the VTGemini mobile app presented in Chapter 4. The self-evaluation is conducted to review the workflow, design, and architecture of the app based on specific quality characteristics. Software Quality Assurance (SQA) is a subjective analysis of the app based on the selected quality indicators we have outlined in this chapter.

5.1 Accuracy

The accuracy of the app is the degree to which the app possesses sufficient transformational, representational, and behavioral correctness.

1. **Validity** is assessed by conducting software validation. Software validation is substantiating that the software possesses sufficient representational and behavioral accuracy. Software validation addresses the question of “Are we building the right software?”

2. **Verity** is assessed by conducting software verification. Software verification is substantiating that the software is transformed from one form of abstraction to another with sufficient accuracy. Software verification addresses the question of “Are we building the software right?”

In order to address these two elements of accuracy, we performed testing and confidence building activities on VTGemini using a myriad of tools. Initially, we utilized built-in tools for Xcode such as debugging and user interface building [Apple, Inc., 2013f]. We utilized Xcode’s Instruments tool to perform performance testing for execution efficiency on energy consumption, storage usage, network usage, and memory usage [Apple, Inc., 2013g]. Additionally, we employed software engineers to assist in functional testing of VTGemini as well as other beta testing activities.

5.2 Functionality

The functionality of a software system is the degree to which the app completely captures all the desired functional modules that need to be present. We conducted the process of Requirements Engineering to identify functional and non-functional requirements as defined by the stakeholders and analyzed by our team. Through the analysis of VTGemini, we satisfactorily addressed and implemented each of the requirements identified by our team.

5.3 Maintainability

The maintainability of a software system is the degree to which the app facilitates corrections, improvements, and changes. Moreover, this is defined by the ability to facilitate an environment for continuous development in the areas of adaptive, corrective, perfective, and preventative maintenance. In order to accomplish the level of maintainability that we desired, we implemented several object-oriented design principles and patterns within VTGemini. By taking advantage of Objective-C (an object-oriented programming language) we were able to provide
an environment that adheres to the Model-View-Controller design pattern [Freeman, Freeman, Sierra, & Bates, 2004, pp. 526-548]. This allowed us to follow the “separation of concerns” principle for different functionality within the app [Freeman, Freeman, Sierra, & Bates, 2004, pp. 9-10].

Additionally, we ensured that the dynamic content was separated from the code-base to create a user interface that was dynamic and easy to maintain. Finally, we maintained a strict observance of inheritance and composition. We utilized inheritance only when we desired certain functionality for a given component and composition only when we desired a reference to a particular object. Composition in iOS development is used quite extensively with outlets to certain user interface components.

### 5.4 Interoperability

The interoperability of a software system is the degree to which the app can exchange data with one or more other software products and be able to use the data that has been exchanged. There are two different types of interoperability that must be addressed: syntactic and semantic. We address the first type, syntactic, of interoperability with the integration of the VMAPush library provided by CNS. The interface and data exchange delegates are well-defined in the VMAPush documentation and allowed to quickly implement a registration and notification services within VTGemiini that was capable of receiving VT Alerts from the Emergency Notification System. Furthermore, we achieve semantic interoperability through the utilization of a common framework that allows us to interpret the incoming JSON notifications; in this case VTGemiini and VMAPush utilize the SBJSON framework for interpreting and rendering JSON objects. With both types of interoperability addressed with respect to VT Alerts and through testing of the communication between the two systems, we can verify that the system properly communicates with CNS and interprets incoming VT Alerts.

### 5.5 Usability

The usability of a software system is the degree to which the app can easily be employed for its intended use. The user experience and usage context includes characteristics such as learnability, effectiveness, intuitiveness, and retainability. We attempted to invoke a positive user experience by utilizing properly designed icons that quickly transfer the type of information the user will expect to encounter. These icons are displayed according to Apple guidelines on size and quality of the icons. Additionally, we maintain a strict color scheme that matches to similar material that is created by OEM in the classroom and on their website. By implementing such a scheme, we can provide a common operational user interface that allows the user to quickly access relevant information across multiple mediums. The effective use of similar icons and color schemes dramatically increases the retainability of the information which is important during an emergency situation.

VTGemiini attempts to provide an easy to use and intuitive interface by simplifying the user interaction to its main components. Information is stored hierarchically based on the level of importance by using expanding table view rows that improve the navigability of VTGemiini. New information is displayed on a new view to wholly encompass and capture the user’s
attention. If a user wishes to quickly access other information within the app, they may activate the sliding view controller to gain direct access to active alerts and guidance material.

5.6 Dependability

The dependability of a software system is the degree to which the app delivers services when requested, delivers services as specified, operates without catastrophic failures, and protects itself against accidental or deliberate intrusion. This was a key feature and requirement for our app from the very beginning of our development. The app was designed to work even when other services within the devices cease to function. We have tested the app without location-based services enabled and without cellular service. The first test essentially prevented us from properly determining the location of the device while displaying active alerts and callboxes. In that case, we simply zoom to the maximum extent of the pins on the map view. The second test ensured that the device continued to function even during an emergency situation where a network connection is unreliable. In both situations, the app continued to function properly and display the correct content to the user.

The security of the app, especially during an emergency notification, is paramount to the dependability. The app itself is a medium for delivery of sensitive VT Alerts that are only issued after a series of protocols are enacted and followed. Therefore, the ability to send Push Notifications only occurs when all parties successfully pass several mutual authentication steps. In our tests, if a single step fails to authenticate with the proper credentials, the app will fail to register for alerts, and, subsequently, fail to receive emergency alerts. Additional protections are taken when registering the device to protect the interests of Virginia Tech by including a Terms and Conditions clause with the notification system. This helps the app protect itself from unlawful actions and intrusions in order to manipulate the system.

5.7 Performance

The performance of a software system is the degree to which the app executes its work in a speedy, efficient, and productive manner. This can include memory usage, storage usage, graphics, network, and power efficiencies. For the purpose of this section, we will review each of these types of tests.

In order to test the memory usage efficiency of VTGemini, we executed the Instruments app provided by Xcode that allowed us to profile app. We were able to analyze and view the memory allocations and memory leaks that were affecting our apps. Fortunately, with the release of the Automatic Resource Count in Xcode 4.2, most of the objects created or referenced in the app are properly cleaned up to avoid potential leaks. Our app proved to perform quite well under the analysis test for Leaks.

The storage usage efficiency of VTGemini was initially a concern when we decided to include all the resources (icons, guides, and configuration files) in the app itself instead of loading them remotely. However, after further analysis of the files themselves, we have concluded that the overall size of the app is reasonable and, in fact, were not as large as we originally anticipated.
Loading content remotely tends to be used by apps that are heavily graphics-intensive (if they are not using OpenGL or Core Graphics) such as games or 3D modeling.

The graphics performance of the app was initially a concern due to the volume of icons and images that are being utilized. Additionally, VTGemini features a fully customized interface for navigation and tool bars throughout. Therefore, we focused on the two main graphics components that handle animations and drawing of objects to the screen. During initial tests of the app on older devices (such as the iPhone 4), we experience a great deal of lag in the scrolling animations of the table and collection views. It was later determined that this particular device was not working properly and every app that included a scroll view experienced the same issue. To display the icons and images in our app we utilized the built in user interface objects for images and image views. However, we implemented the custom navigation and tool bars by accessing the Core Graphics context within our app and drawing the components directly to the screen. The simplicity of our design allowed for this to be the most effective and efficient method for modifying the look and feel of the app. Additionally, it provided a performance boost because we no longer had to redraw the same navigation bar image each time a new view was loaded and pushed onto the navigation controller.

The network efficiency of the app is quite important when we are loading remote content for the hazard guides. We have discussed previously that we include local and remote guides for the user through VTGemini. In order to expedite the delivery of the remote guides and properly display the content to the user, we load and fetch the HTML document from the OEM website and then apply the necessary modifications locally. This allowed us to minimize the potential network degradation that could occur if we were fetching only parts of the HTML document.

The power and energy usage of the app is directly related to the network and graphics performance of VTGemini. Again, we utilized the Energy Usage test within the Instruments analyzer provided by Xcode to determine how much energy a certain process was using and when this occurred. We found that in our tests are app had a minimal impact on the energy usage and consumption on the device. The majority of the impacts that did occur were during the display of remotely loaded content and when the user was performing a search for a particular guide.

Overall, our tests showed that VTGemini performs quite well under the conditions of our tests. We observed that it did not adversely affect the memory usage, storage usage, graphics, network, or power efficiencies of the device. These impacts were also observed during active alerts and each was minimal in the context of the entire device. In fact, simply having the screen turned on with 50% brightness was more detrimental to performance of the device than our app alone.
Chapter 6: Concluding Remarks and Future Research

6.1 Concluding Remarks

Through the analysis of the current landscape of emergency management mobile apps, the app that we designed and implemented exemplifies a niche that has been previously overlooked. VTGemini provides the mechanism to disseminate important emergency information directly to a citizen. The ability to access this information on-demand during an emergency or at their leisure allows citizens to be properly and thoroughly informed about particular emergencies that pertain to the Virginia Tech community. Additionally, the content of emergency guides provides specifically designed directions, checklists, and information for Virginia Tech. While the granularity of the information contained in the app may be seen as an encumbrance by some, the ability to tailor the response of citizens to the specific needs of a single community is invaluable.

The design of VTGemini had to take into account the various branding policies of Virginia Tech. Additionally, the design needed to provide a visual link between the OEM website, guides, and other information with the app itself. The color scheme and image usage within the app must fit these policies and design specifications to create a successful and usable app. The user interface implementation provides a digitized version of OEM emergency guides and the Desk Reference to create a single, unified user experience. This user experience not only included predefined color scheme and designs but also commonly accepted and widely used user interface components, such as a sliding view controller, and the expanding and collapsing table view rows.

Since VTGemini was designed to operate identically on all iOS devices as a universal app, the implementation of the user interface and high-level design was important. In this thesis, we discussed the evolution of the app from a step-by-step guidance app to an emergency information warehouse. We discovered through analysis of the app and the current infrastructure at Virginia Tech that certain concerns impelled the overall architecture of the app. We discussed the benefits of including local and remote content for accessing guides and resources; this includes a discussion about the mutability of the content and its impact on the release schedule of subsequent updates to the app. Additionally, we discussed the importance of an app that had a minimal impact on the network infrastructure during an emergency by way of limiting access to remote content.

With the assistance and collaboration of the Virginia Tech community, Office of Emergency Management, Communications Network Services, and the Virginia Tech Police Department, we were able to create VTGemini to meet the needs and requirements as they were defined and identified. VTGemini includes the ability to disseminate important information through the Emergency Notification System by providing yet another method of communication to citizens. By creating an app that provides clear, concise, and supportive information to citizens during an emergency situation, we believe the app engenders an educated public that can potentially mitigate the extent of collateral damage and secondary incidents while saving lives.
6.2 Future Research

We strived to develop VTGemini with a focus on extensibility and configurability while providing mechanisms to load local and remote content. While we focused deeply on the subject of design and content management, there are several other areas that can be researched and developed to create a feature rich app with respect to emergency notification and education. The most prominent feature that should be included is the ability to accept geographically specific emergency alerts. The current Emergency Notification System at Virginia Tech allows officials to issue alerts based on the campus, such as the Blacksburg main campus or the National Capital Region campus. By allowing citizens to specify their particular campus, the app serves citizens in a more informative and productive way by only notifying the user of relevant alerts.

Additionally, further research and work can be applied to the analysis of the citizen’s current location. For instance, the ability to discern the exact location of the user within a building can be extremely informative in an emergency situation that requires the user to evacuate the building. The app can perform route analysis based on the floor plan of the building, the user’s location and the closest exit. This design can also aid users in accessing and locating Automated External Defibrillators (AEDs) located around campus.

The VTGemini app and supporting systems could be improved with the following:

1. Events and Educational Forum list
2. Extended mapping for all known features and alerts (i.e., buildings, AEDs, and callboxes).
3. Bomb Checklist automation and notification that allows the user to inform the Office of Emergency Management directly.
4. The ability to notify emergency personnel of a threat and the current condition of the user.
5. The ability to “check in” to verify the citizen is alive and well.
References

http://3sidedcube.com/#work/american-red-cross


http://www.emergency.vt.edu/programs/index.html

http://www.police.vt.edu/VTPD_v2.1/selfdefense.html

http://www.emergency.vt.edu/guides/index.html

http://www.branding.unirel.vt.edu/

http://www.unirel.vt.edu/

http://www.wunderground.com/about/data.asp

http://www.wunderground.com/download/index.asp