Project Athena:
A Mobile System Administration Solution

by

Andrew Smiley

Submitted to
the Faculty of the School of Information Technology
in Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science
in Information Technology

© Copyright Andrew Smiley

The author grants to the School of Information Technology permission
to reproduce and distribute copies of this document in whole or in part.

[Signature]
04/25/2015
Date

[Signature]
04/25/2015
Date
University of Cincinnati
College of
Education, Criminal Justice, and Human Services
April 2015

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>BACKGROUND</td>
<td>4</td>
</tr>
<tr>
<td>STATEMENT OF NEED</td>
<td>6</td>
</tr>
<tr>
<td>FUNCTIONALITIES</td>
<td>7</td>
</tr>
<tr>
<td>USER PROFILES</td>
<td>7</td>
</tr>
<tr>
<td>BUDGET</td>
<td></td>
</tr>
<tr>
<td>TECHNICAL REQUIREMENTS</td>
<td>9</td>
</tr>
<tr>
<td>TECHNICAL CHALLENGES</td>
<td>10</td>
</tr>
<tr>
<td>IMPLEMENTATION AT A GLANCE</td>
<td>10</td>
</tr>
<tr>
<td>GANTT CHARTS</td>
<td>23</td>
</tr>
<tr>
<td>SYSTEMS DIAGRAM</td>
<td>24</td>
</tr>
<tr>
<td>USE CASE DIAGRAM</td>
<td>25</td>
</tr>
<tr>
<td>TESTING/EVALUATION</td>
<td>26</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>29</td>
</tr>
<tr>
<td>SCREENSHOTS</td>
<td>31</td>
</tr>
<tr>
<td>SOURCES</td>
<td>32</td>
</tr>
</tbody>
</table>
Abstract

Today, the workload of the system administrator is ever increasing. The number of systems to manage is increasing, along with the number of requirements to manage said systems. Our mobile application seeks to alleviate this issue. With our mobile application, we were seeking to offer the user the same flexibility and control of systems they might have on a PC, in addition to integrating voice recognition technology with word recognition improvement algorithms. Our hypothesis was two-part: 1. Multi-system management can be achieved through ad-hock reflection and 2. Word recognition can be improved by algorithmically calculating possible alternatives for single word. Our findings indicated that multi-system management is possible through ad-hoc reflection. Additionally, we found that word recognition accuracy can be noticeably improved by preemptively calculating possible words or phrases voice recognition software may mistake for the specified word.
Background

Among the many ever increasing advances in the computing world is the topic of cloud computing, or virtualization. According to the 2014 Cisco GCI (Global Cloud Index) Report, it is estimated that 76% of all data center traffic will come from the cloud by 2018. Additionally, last year, we saw the first year in which the majority of workloads were contained in the Cloud, again, according to the Cisco GCI Report. With this increase in Cloud-based services, comes an increase of potential means for accessing and administering resources. For example, with a physical server, a user's means of accessing its resources are limited to having physical access to the machine itself; whereas with cloud based services, a user has multiple means of accessing computing resources, such as web interfaces (such as Amazon Web Services), SSH terminal connections, Remote Desktop Protocol (RDP) or Virtual Network Computing (VNC), etc. Often, systems offer multiple mediums for connections and management, such as Amazon Web Services' EC2 cloud servers, which offer both a Secure Shell connection option, as well as a web management console. Many of the aforementioned management mediums are not limited to Cloud-based resources, such as RPD, VNC or SSH, but can also be configured to work with physical resources as well. For example, a physical Linux system can be configured to accept connections via the Secure Shell protocol or VNC, potentially both at the same time. We knew from experience, that often, Unix-based systems, primarily Linux, serve as database servers. Of those database servers, it's known that MySQL makes up an estimated 56% of the open source database market1. Therefore, it is safe to say that
MySQL makes up a large percentage of database, thus making it an important topic in the conversation of system administration.

With the information we had gathered regarding system administration, we knew that we wanted to develop a tool to ease give more power to the system administrator. We already knew that the market for PC-based system administration systems was already well tapped, thus we investigated alternative mediums for system administration. Our decision to develop a system for mobile devices was based upon the following information: We knew that the market for smartphones was very vast, roughly around 4 billion², which gave us a very good indicator of the popularity of smart phones. There was an assumption that system administration applications already exist for smartphones, of which our assumption was correct. However, we found that the existing systems on smart phones for system administration were rather lacking. Some applications that were analyzed included Server Auditor, MySQL Mobile Database Client, IT Manager and several other applications, which shall remain nameless. The majority of systems for mobile devices were lacking in functionality conducive to a mobile medium. Our findings showed that the large number of applications focused primarily on offering the user only console sessions, which often proved tedious to utilize, requiring the user to remember system commands and syntax, which again, proved to be very tedious on mobile devices. Figure 1 shows an example of console sessions on mobile devices. This was the problem we desired to provide a solution to with Project Athena. More formally our goal was to develop a system for mobile devices, which would simplify the execution of tasks on remote systems for the end user. With
this goal, we developed a two-tiered approach for achieving the aforementioned goal. In
the first tier, we wanted to develop a graphical user interface (GUI) for executing routine
tasks, such as rebooting a system, managing software on a remote system, etc with
dynamic feedback from the remote system. In the second tier, we wanted to develop a
voice command functionality, which would allow the user to complete more complex
tasks, such as custom terminal commands. On top of the voice command functionality,
we knew that we would need to develop an alternative approach to recognizing words
that may not be picked up by a voice recognition engine, such as command names,
software names, etc. Finally, we wished to do all of the aforementioned tasks with as
little system overhead as possible.

Statement of Need

In this day and age, as more services are being moved to ‘the cloud’, as well as
the number of LAN based services increasing; there is a lack of mobile management
applications for interacting with these cloud-based services. Although some solutions do
exist, there is no simplified mobile application for system administration. This is where
the Server Manager mobile application comes into play. The Server Manager mobile
application offers a simple, intuitive interface for managing MySQL and Linux-based
systems.
Functionalities

The mobile application will have the functionalities:

- Complete system control using a user interface designed for mobile devices, specifically iOS devices
- Management of MySQL systems utilizing a user interface designed for mobile devices, specifically iOS devices
- Speech recognition for managing systems
- Implementation of various other physical analytics

User Profiles

<table>
<thead>
<tr>
<th>Potential Users:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● System Administrators</td>
</tr>
<tr>
<td>● Database Administrators</td>
</tr>
<tr>
<td>● Freelance Developers</td>
</tr>
</tbody>
</table>

Software and Interface Experience:
The user should have a basic understanding of Linux or Windows system management as well as system services. In addition, the user should have a level of understanding of MySQL databases.

<table>
<thead>
<tr>
<th>Experience with Similar Applications:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Windows Remote Desktop</td>
</tr>
<tr>
<td>● SSH Sessions</td>
</tr>
<tr>
<td>● MySQL Workbench or similar applications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Experience:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Managing system services</td>
</tr>
<tr>
<td>● Managing users</td>
</tr>
<tr>
<td>● Managing installed software</td>
</tr>
<tr>
<td>● CRUD operations in MySQL</td>
</tr>
<tr>
<td>● Schema and User Management in MySQL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequency of Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ideally, this application would be used daily by any number of the aforementioned users.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key Interface Design Requirements that the Profile Suggests:</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Server overview views to allow the user to quickly manage system services, install software, etc.</td>
</tr>
<tr>
<td>● Terminal views to allow for immediate command execution</td>
</tr>
<tr>
<td>● For Windows systems, a VNC-based view to allow users the same experience as they would be allotted in a Remote Desktop application</td>
</tr>
</tbody>
</table>
- Views for listing schemas, tables and data
- Views for execution of custom written queries

**Budget**

Server Manager, the mobile system administration application, is an open sourced application, which is being developed on a limited budget. Estimated developing costs are estimated as the following:

- Virtualization Software: $70
- Membership to iTunes Developer Center: $100 (annually)
- Cost of development (Freelance rate): ($30.00/hour) X (20 hours per week) X (26 weeks) = $15600

**Estimated Final Cost of Production:** $15770

**Technical Requirements**

There are several technical requirements to implementing this project. This project will require the virtualization of several Linux distributions, including, but not
limited to Ubuntu, CentOS, Slackware Linux, Debian Linux, RedHat Linux, Mint Linux. This project will require the use of the following software: Xcode and VMWare (fusion or workstation). In addition, the development of this project will require the use of an Apple iPhone and iPad running iOS 8.

**Technical Challenges**

There are several technical challenges associated with this project. They are as follows:

1. Handling session connection issues
2. Generically handling server operating systems in-app, i.e. different systems use different system applications for management (yum vs apt-get for example)
3. Properly handling password storage
4. Outputting connection status information to the end-user
5. Simplifying system management into an easy-to-use and intuitive GUI
6. Utilizing a MySQL Library for iOS devices (to which none exist)
7. Implementing speech recognition

**Implementation At a Glance**
**Controlling Remote Systems:** Our approach to controlling systems took several iterations before we were able to establish an effective methodology for connecting to remote systems. Initially, we investigated the concept of remote scripting, in an effort to avoid actually connecting to the remote system via Secure Shell. However, this approach proved problematic, as it requires a user to preemptively create scripts to execute remotely and additionally, this approach still requires a Secure Shell connection to a server, even if temporarily. We realized that ultimately we would be required to establish a Secure Shell session to a remote system, which we elected to use to our advantage. Ultimately, we opted to utilize a Secure Shell SDK for iOS, written in Objective-C, NMSSH\(^3\). Essentially, this is an Objective-C implementation of libssh2, allowing us to open Secure Shell sessions, and communicate with a remote server, sending commands and receiving responses. This is the foundation for our system of remote system management. In essence, an end user is offered an action, typically a button click, selection, etc, which upon activation, triggers an event which sends a corresponding command to a remote system; we then process the feedback from the server, generally parsing with REGEX and giving the end-user feedback about the result of the action;

**Console Sessions:** Although our proposed application emphasizes simplicity for the user by breaking down system-related tasks to GUI based actions, we also offer the end-user another functionality they would have on a PC client, console (terminal) sessions on remote systems. These console sessions are achieved though the use of a series of Pseudo-terminals\(^4\). To begin, a terminal in its simplest form is an electronic hardware
device which is used for entering data into and displaying data out of a computing system. A pseudo-terminal is essentially a device which emulates a physical terminal, allowing a user the same functionality as a hardware terminal as a piece of software. Pseudo terminals function by utilizing a communication channel that emulates a hardware terminal. One end of the channel is called the master side or master pseudo-terminal device, the other side is called the slave side. Data written to the master side is received by the slave side as if it was the result of a user typing at an ordinary terminal, and data written to the slave side is sent to the master side as if it was written on an ordinary terminal\(^5\). Secure Shell is an example of a Pseudo Terminal, differentiating from other Pseudo Terminals, such as Xterm\(^6\), in that Secure Shell communicates with a remote system, rather than a local system. Our proposed application adds a second layer Pseudo Terminal on top of the Secure Shell session, allowing a user to enter commands like they would in a Secure Shell session, basic Pseudo Terminal or terminal, and receive feedback as they would in the aforementioned mediums. We call the terminal functionality in our proposed application a second-layer Pseudo Terminal in the way that this functionality works. The application begins initiates communication with a remote system by requesting a Secure Shell Pseudo Terminal. If the request is successful, we draw a terminal for the user to utilize, just as they would on a PC. The end user then has the ability to enter data and execute programs on the remote system, as well as view the output from the remote system. These data inputs go through the Secure Shell Pseudo Terminal first, before being sent to the physical or virtual remote system. In the same way, the returning data comes first through the Secure Shell Pseudo terminal running on
our application, which is then read by our application and displayed back to the user, giving the illusion that the end user is directly communicating with the remote system. Figure 2. shows this flow of data.

**Large Data Output:** As previously stated, a challenge we face in designing the proposed application is the challenge of the size of the screen of the device. Apple iPhone 5 devices have screen sizes of 1136x640 pixels and in our Pseudo terminal; we utilize a font size of 14, which translates to a line height of roughly 28 pixels per line. Accounting for the onscreen keyboard, which is 224 points (1 point = 4 pixels), gives us a display size of roughly 240 pixels. This gives us a roughly 8, or 38 without the onscreen keyboard, displayable lines of output. This becomes a challenge when the output from the Secure Shell Pseudo Terminal is greater than 8 lines, which can happen very often. Our proposed solution is to dynamically re-draw resize the area of the canvas the text is displayed on and implement a UIScrollView to allow the user to scroll through data output that has either (a.) extended beyond 8 lines, or (b.) exceeded the height of the device screen. Another challenge with a limited screen size is displaying text lines with a length greater than 640 pixels. Our solution to this challenge was simple. By calculating the length of the line, we determine whether it will extend beyond the width of the screen. If the length of the line will extend beyond the width of the screen, we find the last word in the line that does not extend beyond the width of the screen and insert a line break immediately following the word. When then iteratively draw the remaining lines, maximizing the amount of text per line, while not exceeding the width of the screen. One caveat to be accounted for is a scenario in which a line break is inserted after a word and
the following word or series of words does not come within 75% of the width of the screen, before another line break exists in the text.

**Process Management:** Another piece of functionality that we wanted to offer our users was the ability to control processes on remote systems. The vision of how this functionality would work was simple, but effective; A user would select the process management console for a particular server and would then be presented a list of services and their statuses of which they could manage in a similar form to how they would on a PC, by starting, stopping, and restarting the particular service. We postulated several means for building this particular functionality. Initially, we researched potential commands that would allow us to list all of the services available to a user. One particular command that we found moderately successful was `service --status-all`. While this command was successful, it had one caveat; this command only listed the running services on a Unix system. This was not ideal, as we wanted to offer the user the ability to list services that were both running and stopped, that would allow them to start new processes. With the lack of success of using commands to list the services available to a user, we were required to research alternative options. In the search, we discovered that all of the services available to the user (as they are simply scripts) were stored in the `/etc/init.d/` directory. We attempted to display the contents of the `/etc/init.d` directory, as the contents of `/etc/init.d/` are shell scripts that respond to start, stop, restart, and (when supported) reload commands to manage a particular service and found that we were highly successful in this approach. We were successfully able to display all of the
services available to a user and allowed the user to control the services with the touch of an on-screen button.

**Software Management:** When conceptualizing features that were available to the PC client user, we spoke to several system administrators in regard to what they would like to see. A resounding response was that they would like to manage software packages from a mobile device. The actions that they would like to see include the ability to install new software, uninstall software packages and update software packages in a simple, easy to use manner. One challenge that we discovered when building this piece of functionality was determining which particular package manager was appropriate for a particular system. We initially researched using system commands to determine which package manager was most appropriate for use on a particular system, however, we found that we could not successfully determine which package manager would be most appropriate. To overcome this challenge, this required a reevaluation of how we were creating system connections. Our solution was to gather additional information when the user created a new system connection. Using iOS CoreData, we created system objects in the iOS device's local storage, which would both a system type with a name, and the name of the preferred package manager for that particular system. In that way, when a user creates a new system connection, we will know which default package manager to use for that connection session. To offer flexibility, we opted to offer the user two forms of package management; a simple package management functionality, which would use the default package manager for the system, and a manual package manager, which
would prompt the user to chose their preferred package manager. Regarding installing new software, we could query the repositories of the preferred package manager to generate a list of software packages available for installation, of which a user could select a package for installation. In the same manner, we were able to create the functionality to allow a user to remove and update software packages as well.

**MySQL Connectivity:** While our application emphasizes remote connectivity to systems (servers), we would be behooved not to incorporate database management, as many Linux systems function primarily as database servers, which exist as servers, running a database service. As previously stated, we chose to offer support for connectivity to MySQL services, as MySQL encompasses more than half of the open-source database market. In keeping with the mindset of allowing a user to execute the same level of tasks on a mobile client as they would on a PC client, we were required to incorporate the following pieces of functionality, which include but are not limited to schema management, table management, CRUD operations and direct SQL execution.

**Lack of MySQL Libraries for iOS:** While the implementation of these pieces of functionality would prove to be straightforward, establishing a connection and communicating with a MySQL service running on a remote server would prove to be more challenging than anticipated. Initially, we hypothesized that establishing a connection and communicating with a MySQL service running on a remote server would be simple and could be done natively in Objective-C. Further research proved that due to
the API-centric nature of mobile applications, Objective-C did not offer any out-of-the-box solutions for establishing this type of communication. Although it would be possible to create such a library in-house, we were optimistic that such a library had previously been created and could be implemented in our application and we believed that developing such a library would extend beyond the scope of the proposed application. However, our optimism proved incorrect, as no MySQL communication libraries existed in Objective-C for iOS deployment. It is worth noting that such a library does exist, MySQL Cocoa, however it exists solely for OS X application development, as it utilizes frameworks available only to OS X applications. We reached a crossroads in whether to drop the idea of MySQL support for the proposed application, or continue researching alternative means for establishing a connection and communicating with a MySQL service running on a remote server. Using the knowledge that Objective-C can compile pure C, written inline, we pursued C libraries written for communicating with MySQL and found a successful match in MySQL connector for C, published by MySQL. As this was a library that had to be imported into our application, we had to generate a static library file capable of being compiled and used with iOS from MySQL Connector for C. This task proved challenging at first, but we found success by cross compiling for arm6, arm7 and i386 processors. This was achieved using Cmake, an open-source build tool, changing several of the C headers to become compatible with our desired processors, compiling the libraries for each desired architecture, combining them into a single static library file using Lipo and then importing into our application along with the C header files. This
process allowed us to utilize the functionality of the C library, inside of our Objective-C application, thus allowing us to manipulate MySQL services running on remote servers in a similar fashion to our aforementioned control process of solely remote systems.

**Voice Recognition and Word Recognition Algorithm**

As previously stated, in an effort to further simplify the tasks of the user, we decided to implement voice recognition to allow the user to execute certain tasks on a remote system. In order to make this task a possibility, we implemented the Nuance SDK for iOS. Nuance is a top company in voice recognition, the same company behind Dragon Dictation, among other products. Our goal was not to reinvent the wheel by creating our own voice recognition library, but rather utilize existing libraries to accomplish our larger goal. Using the existing stored system(s) information; we generated a series of commands for each stored system and stored them in a helper class before any voice input is taken. The Nuance SDK processes voice input and returns it as a string, using a series of delegate functions. When this has been done, we iterate through each instance of the helper class and then iterate through each potential voice command. However, while we were successful in using the Nuance voice recognition library, we discovered a challenge when recognizing words that were not necessarily existent in the English dictionary. For example, a specific challenge we discovered was with the word "Ubuntu". While the phrase, "shutdown Ubuntu" is straightforward to an English speaker; we discovered a caveat with the Nuance SDK. We discovered that a word such as "Ubuntu" had the potential to be recognized not as a single word, but rather a series of smaller words, like
"you bun too", "you bunt ew", etc. This became more evident when beginning to work with services such as nginx, which is pronounced entirely different from how it is spelled, thus leading to the Nuance SDK failing to ever generate a command sequence we would recognize. This led us to hypothesize methods in which we could improve this word recognition caveat. We began by postulating how to identify possible alternative words or phrases for our keyword(s). We hypothesized that we could break a keyword down into its individual letters and generate alternative words or phrases iteratively based upon each letter or a series of letters. Research into this hypothesis proved that this was in fact a known phenomenon, known as grapheme-to-phoneme mapping, or mapping letters or letter series to a particular phonetic sound, e.g. "u" to the sound "y+oo", "y+ew", etc. We began researching both the topics of both grapheme-to-phoneme mapping and phoneme-to-grapheme mapping to better understand how we could algorithmically accomplish this task to generate our alternative words. Using the work of Maximilian Bisani and Hermann Ney as a starting point, we knew that there were multiple approaches to accomplishing the task of grapheme-to-phoneme mapping. These approaches included, a dictionary-based approach, the simplest approach to implement, an approach that involves a dictionary lookup based upon the selected grapheme; a rule-based approach, which involves utilizing linguistic rules applied to graphemes to generate a related phoneme; finally, Bisani and Ney suggested a third methodology for grapheme-to-phoneme conversion, which is a data-driven approach based on the idea that given enough examples it should be possible to predict the pronunciation of unseen words purely by analogy. Ultimately, we elected to use a combination of both a
dictionary and rule-based approach to match graphemes to phonetic sounds to generate possible alternative words or phrases.

To begin our algorithmic approach, we knew we needed to generate a dictionary file which our algorithm could rely on to find potential alternative word matches for a specific grapheme or series of graphemes as well as a set of rules we could apply to letters or series of letters. To create the rule set, we were simply able to research grapheme-to-phoneme mapping and find a rudimentary grapheme-to-phoneme list for the English language and correspond this to a file which we would later parse. However, generating the dictionary was rather more involved. To do so, we began by downloading a text file containing every word in the English language. We then wrote a simple Python script, utilizing the rule set we had created, to generate a file containing every word in the English language and each potential phonetic spelling of the word.

When completed, the dictionary contained lines like this:

\[ y + oo = you \]

\[ y + ew = you \]

Etc.

Once we had generated our dictionary file, we were able to begin designing our algorithmic approach. At its core, our algorithm states that for all phonemes in grapheme-phoneme set \( A \), there exists a match, \( m \), in dictionary, \( D \), where phonetic sound \( p \) in \( A \) is equal to \( m \). Additionally, our algorithm states that each letter or series of letters \( l \), in word \( W \), there exists a match \( m \) in rule set \( R \) such that \( l \) is equal \( m \). Using those existential statements, we were able to establish an algorithmic methodology for
generating alternative words or phrases based upon grapheme. As we mentioned before, we utilized a helper class to store the voice commands available to a particular system; we decided to add to this class a list of alternative words or phrases for our keyword, to be added after the algorithm has been implemented. Essentially, our algorithm takes the following steps to generate a list of alternative words:

1. Create two multidimensional arrays to store all potential word matches and phonemes.
2. Using a specified keyword, split into an array containing each letter of the particular word.
3. After iterating through the word, we then begin iterating through each array of phonetic sounds, finding each word in our dictionary whose phonetic makeup is equivalent to the phonetic iteration in our array. When the match(es) are found, add them to the multidimensional array we are storing potential matches in.
4. When we've established all the possible word matches, we iterate through all the possible combinations and generate alternative command strings we can use when/if no matching command strings were found by the Nuance SDK.

Exceptions: this approach works well when the selected keyword does not contain words inside of itself, e.g. "nginx", which does not contain any sub-words. However, when a word such as "Ubuntu" contains sub words; the word "Ubuntu" contains the sub word "bun". The base algorithm does not pick up on situations such as this because, although "bun" is a word in the English language, it is not a grapheme inside of our ruleset. This has the potential to contaminate potential alternative words or phrases as the algorithm
generates alternative words or phrases based solely upon individual grapheme or letter, thus when creating phrases, the initial algorithm has the tendency to generate more words alternative in a phrase than would be said. For example, for the word "Ubuntu", the algorithm in its initial state would generate the alternative phrase "You be on to", rather than the more accurate "You bun to", which is more likely to be picked up by the Nuance SDK. Therefore, we were required to make a slight adjustment to the algorithm in order to account for this discrepancy. This requires us to utilize a second dictionary file, containing only the words of the English language, not their phonetic makeup as well.

The improved algorithm takes the following steps to ensure the most accurate list of alternative words/phrases:

1. Create two multidimensional arrays to store all potential word matches and phonemes.
2. Using a specified keyword, split into an array containing each letter of the particular word.
3. For each letter n at position, determine if the combination of n through n+word length creates a grapheme found in our ruleset or a word in the dictionary. If a word in the dictionary is not found by n+word length, begin searching for graphemes, starting at n through n+word length, accounting for cases where it does not. For example, if letters in range, n-n+3 create a grapheme found in our ruleset, but letters in range n-n+4 do not, use the range n-n+3 and add possible phonemes to our array of phonemes, setting the new index of n = n+4. Repeat through the length of the word. However, if a dictionary word is formed form
from letter range n - n+x, where x is the current iterative index, add that word to our list of potential alternative word combinations and update n = (n+x)+1.

4. After iterating through the word, we then begin iterating through each array of phonetic sounds, finding each word in our dictionary whose phonetic makeup is equivalent to the phonetic iteration in our array. When the match(es) are found, add them to the multidimensional array we are storing potential matches in.

5. When we've established all the possible word matches, we iterate through all the possible combinations and generate alternative command strings we can use when/if no matching command strings were found by the Nuance SDK.

Gantt Charts
Use Case Diagram
Testing and Evaluation

When we began the testing portion, there were multiple criteria to meet for what we considered a reasonable application for a mobile application. These criteria were as follows: memory usage, CPU usage and battery usage.

For testing, we utilized an iPhone 5 running Apple's iOS 8.1.2 and an iPhone 4 running Apple's iOS 7.1.2. The iPhone 5 was released in 2012 and features a dual-core 1.3 GHz Swift (ARM v7-based) CPU, 1 GB of DDR2 RAM and a non-removable Li-Po 1440 mAh battery (5.45 Wh). The iPhone 4 was released in 2010 and features a 1 GHz Cortex-A8 CPU, 512 MB RAM and a Non-removable Li-Po 1420 mAh battery.

**Memory Usage:** On the iPhone 5, we initially estimated that the mobile application would average roughly 10mb of memory when in use. However, what we found was quite to the contrary. When idling with no server connections open, the application used only 3.8mb of RAM, not nearly half of what we anticipated. When idling with a connection to a server open, we saw a very small spike in the amount of RAM utilized, roughly 4.7mb.

One area we anticipated higher than normal memory usage was in the voice recognition functionality of the application. Due to the high workload the application
executes during the loading stage, such as calculating alternative words, loading word
dictionaries, etc, we anticipated at least 10% memory usage. Surprisingly during this
stage, the mobile application's highest memory spike was merely to 30.6mb of memory,
which is far less than 10% of memory usage. When idling in the voice recognition
portion of the application, the application averaged roughly 15.8mb of RAM used.

After seeing the results from the iPhone 5, we expected to see similar memory
usages on the iPhone 4. Contrary to our hypothesis though, we found that the memory
usage on the iPhone 4 differed somewhat from the memory usage on the iPhone 5. When
idling with no remote system connections, we found that the application used only 2.9mb
of memory, about 0.1% less memory than the iPhone 5. With the application idling with a
remote system connection open, we saw application memory usage of roughly 3.8mb on
the iPhone 4, which again, was nearly 0.01% less than the memory used by the iPhone 5.
As with the iPhone 5, we expected a higher than normal average memory usage when
using the voice control functionality, but a lower memory usage when compared to the
iPhone 5 in regard to memory. However, contrary to our expectations, we saw a higher
spike in memory, up to 33.2MB and a 1% greater percentage of memory used when
idling, around 25.6MB.

**CPU Usage:** When testing on the iPhone 5, we anticipated fairly high levels of CPU
usage, roughly 5-10% at a minimum. However, the test results proved to be contrary to
our initial hypothesis. On application launch, the application reaches 12% CPU usage,
however, when idling with no remote system connections, the application used 0% of the
system CPU. When actively communicating with a remote system, the application averaged <5% of total CPU usage, and again, when idle with a remote system connection, the application utilized 0% of total CPU usage.

Once again, the edge case regarding CPU usage was the voice recognition functionality of the application. When loading the voice recognition functionality of the application, we saw a large CPU spike of up to 50%, however, that number decreased back down to 0% CPU usage once the loading had ceased.

As with testing the memory usage of the application, we suspected that the CPU usage of the iPhone 4 would be similar to the CPU usage of the iPhone 5. Again, the results we saw were somewhat to the contrary to the results we had hypothesized. When launching the application, we saw that the CPU usage of the application spiked to 40%, which was nearly four times the CPU spike we saw when launching the application on the iPhone 5. When the application instantiated a connection to a remote server, we saw the same CPU usage on the iPhone 4 as we did on the iPhone 5, which was roughly 12%.

Arguably, the worst spike in CPU usage we saw was when using the voice recognition functionality. When launching the voice recognition functionality, we saw a spike in CPU usage of 100% of CPU used. Surprisingly, this did not cause any instability in the application; however, this was still less than ideal.

**Battery Usage**: On average, the expected battery usage of the iPhone 5 is 225 hours of standby on 3G networks, whereas the iPhone 4 the expected battery usage is up to 300 hours of standby on 3G networks. Due to the required libraries of the application, we did
not run any testing analytics on simulators. To evaluate the memory usage, CPU usage, and network usage of the mobile devices, we used the analytic tools built into Xcode 6.1. For our battery testing, we did not utilize the built-in analytical tools available in Xcode, but rather relied on observations over periods of time. To create a control, we monitored the battery usage on the iOS devices when no background applications were running, with the iOS device set at 50% brightness, connected to a wireless network. During this control observation, we noted that the iPhone 5 utilized 0% battery over a ten-minute interval.

When testing the application, we continued to utilize ten minute intervals using the following scenarios, application idle (no system connections), application idle, (with system connections), application idle (MySQL connection activated), application idle (voice recognition screen), application active (system management), application active (MySQL management), application active (voice control). Surprisingly, we saw very little battery usage for all idle states over ten minute interval, averaging <1%. However, when the application became active, we did observe slightly different results. When active performing system management we saw a battery usage of 1-2% over a ten-minute interval. Alternatively, when active and performing voice recognition functionality, we say a battery usage of 3-5%, which was rather concerning, as this could wear down a user's battery over the course of a day. When testing with the iPhone 4, we saw nearly identical
Conclusion

We developed a fully functional remote system management client for mobile devices, offering the user the same functionality that they would be offered on a PC client. To allow users control, we wired remote system commands to click-based actions. To incorporate terminal functionality, we created a 2-layer Pseudo terminal system. We incorporated a fully functional MySQL client with similar capabilities that a user would have on a PC client. In simplification of a user's tasks, we incorporated voice recognition, which we were able to improve the accuracy of by using a grapheme-to-phoneme algorithm. Our algorithm successfully increased the accuracy of keyword recognition using a combination of rule-based and dictionary lookup techniques. Our prototype showed that it was possible to remotely manage a system with a graphical user interface and a minimal knowledge of system administration.
Screenshots
account-plugin-aim
account-plugin-facebook
account-plugin-flickr
account-plugin-google
account-plugin-jabber
account-plugin-salut
account-plugin-twitter
account-plugin-windows-live
account-plugin-yahoo
apparmor
mysql

Host
192.168.1.114:22
Operating System
Ubuntu
Username
andrew
Edit Server
Manage
System Overview
Delete

Command

Execute

Review Command

Start Listening

Stop Listening
Sources

3. https://github.com/Lejdborg/NMSSH
6. http://invisible-island.net/xterm/
11. https://hal.archives-ouvertes.fr/hal-00499203/document