Mobile Application for Virtual Communication Assistance

GRADUATE PROJECT REPORT

Submitted to the Faculty of
The School of Engineering & Computing Sciences
Texas A&M University-Corpus Christi
Corpus Christi, TX

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

by

Parthasarathy Krishnamoorthy
Summer 2013

Committee Members

Dr. Ahmed M. Mahdy
Committee Chairperson

Dr. Ajay K. Katangur
Committee Member

Dr. Scott A. King
Committee Member
The objective of this project is to develop a real time application to identify the email reply patterns. This application exploits the ability of Natural Language Processing (NLP) to identify patterns based on email conversation logs. Stanford Parser is used for parsing the conversations to identify the parts of speech of all the tokens in every message. In addition, Cosine similarity algorithm is used for measuring the similarity among the messages and Levenshtein algorithms is used for measuring the difference in distance between the messages. Based on the result got from the algorithms, a reply pattern is identified. In addition the application manages calendar entries for appointment request which are received through emails and reschedules overlapping appointments based on the sender’s priority. The application is developed on the Android platform, considering the usability and APIs it supports.
# TABLE OF CONTENTS

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

Table of Contents .......................................................................................... iii

List of Figures ............................................................................................... v

List of Tables ................................................................................................. vi

List of Code ................................................................................................... viii

1. Background and Rationale ......................................................................... 1

   1.1 Natural Language Processing .............................................................. 1

   1.1.1 Machine Translation ................................................................. 1

   1.1.2 Text Summarization .................................................................. 2

   1.1.3 Dialogue System ....................................................................... 2

   1.2 Android Operating System ............................................................. 2

   1.3 Virtual ME ................................................................................... 2

2. Virtual Communication Assistance ........................................................... 4

3. Requirements ............................................................................................. 6

   3.1 Plain English Description .................................................................. 6

   3.2 Functional Requirements .................................................................. 7

4. Design ........................................................................................................ 9

   4.1 Client Design ................................................................................. 9

   4.2 Server Design ................................................................................ 10

   4.3 Use Case Diagram ......................................................................... 12

   4.4 Class Diagram ............................................................................... 14

   4.5 Activity Diagram ............................................................................ 16

   4.6 Sequence Diagram .......................................................................... 18
4.7 User Interface........................................................................................................20

5. Implementation ........................................................................................................22
   5.1 Assumptions.......................................................................................................22
   5.2 Client side ..........................................................................................................22
   5.3 Server side .........................................................................................................24
   5.4 Limitation ..........................................................................................................26

6. Testing, Results and Evaluation .............................................................................31

7. Conclusion and Future Work ..................................................................................36

8. Bibliography and References ..................................................................................39

9. Appendix A: Textual Description for Use Cases .........................................................40

10. Appendix B: Code Snippets .....................................................................................45
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Modular Block Diagram</td>
<td>4</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Level 1: Main Use Case Diagram</td>
<td>12</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Level 2: Use Case for Fetch email</td>
<td>13</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Level 2: Use Case for Manage Calendar</td>
<td>13</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Server Class Diagram</td>
<td>14</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Client Class Diagram</td>
<td>15</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Server Activity Diagram</td>
<td>16</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Client Activity Diagram</td>
<td>17</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Sequence Diagram for Fetch email Use Case</td>
<td>18</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Sequence Diagram for Manage calendar Use Case</td>
<td>19</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Client User Interface: Start Up View</td>
<td>21</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Client User Interface: Add Contact View</td>
<td>21</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Client User Interface: Show Contacts View</td>
<td>21</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Date Computed by the Parser</td>
<td>23</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Implicit Intent to Add Event</td>
<td>25</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Message before Cleaning</td>
<td>26</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Message after Cleaning</td>
<td>26</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Message Tokens Tagged with POS</td>
<td>27</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Final Dataset Collected</td>
<td>28</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Distance Computed Using Levenshtein Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Cosine Similarity between Messages</td>
<td>32</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Reply Pattern Plotted as Graph</td>
<td>33</td>
</tr>
</tbody>
</table>
Figure 6.2.3 Custom Priority Contact List ..................................................33
Figure 6.2.4 Calendar with no Event .............................................................34
Figure 6.2.5 Event Created at the Requested Time ......................................34
Figure 6.2.6 Calendar with Event ...............................................................34
Figure 6.2.7 Event Rescheduled based on Priority ......................................34
Figure 6.3.1 Assumed Possible Replies .......................................................35
Figure 6.3.2 Result Comparison based on POS Frequency ............................35
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Parts of Speech and its Description</td>
<td>11</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Dataset created by processing emails</td>
<td>28</td>
</tr>
</tbody>
</table>
LIST OF CODE

5.1.1 Code Snippet for Adding Event to Calendar ...........................................45
5.1.2 Code Snippet for Fetching Data from Custom Database ..................................45
5.1.3 Code Snippet for Generating Graph ..............................................................46
5.2.1 Code Snippet for Server Connection ...............................................................46
5.2.2 Code Snippet for Fetching Email from Mail Server ...........................................47
5.2.3 Code Snippet for POS Tagging ....................................................................47
5.2.4 Code Snippet for Cosine Similarity using Levenshtein Algorithm ....................48
1. BACKGROUND AND RATIONALE

Many researches are being conducted to improve the standard of technology. Researches on Artificial Intelligence such as Apple’s SIRI, Google Now and so on have helped users to perform tasks more efficiently in lesser time. This improved speed and performance lets users to increase their productivity and efficiency. Natural Language Processing (NLP), machine learning, neural networks and so on have been the key area of interest in Artificial Intelligence. Natural Language Processing provide many approaches to collect and analyze data from various sources to improve Artificial Intelligence.

1.1 Natural Language Processing

Natural Language Processing (NLP) is a field in computer science where computers use natural language as input and output. Natural language processes is one among the most advanced concepts in Artificial Intelligence. Projects such as SIRI and Google Now are the most popular example of Natural Language Processing. There are a few basic problems which NLP addresses. NLP provides good platform for research [1]. Following are the popular NLP techniques,

1.1.1 Machine Translation

Natural Language Processing has been widely used for machine translation. The ability of an application to process and display an input (written in one language) in multiple languages is called machine translation. Google translate and Multilingual Online Translation (MOLTO) is popular machine translation applications.
1.1.2 Text Summarization

A Text summarization is another common challenge in Natural Language Processing. The challenge is to produce a summarized passage or message based on multiple inputs from various sources. The NLP engine processes multiple input resources regarding the same or similar subject and produces a consolidated or summarized report [3].

1.1.3 Dialogue Systems

Chatbots or chatterbots are the most common and popular NLP dialogue systems available. Most chatterbots scan keywords within the input and find a reply with the most matching keywords from a database. There are very few chatbots which utilize proper NLP to produce results.

1.2 Android Operating System

Mobile platforms provide easy and efficient use of technologies required for day to day use. Android having open SDK’s and APIs provides to programmers easy and abundant opportunity for development. Though Android fails to support advanced scripts such as AppleScript, it does provide numerous third party libraries to the job for free. In addition it is very easy to test Android application. Unlike Apple, it is easy to install unpublished Android application. Also, according to Forbes, Android devices held 51% and currently hold 70% of market share in Smartphone industry. Considering this growth, the project is implemented on the Android platform, to reach as many users as possible.

1.3 The Big Picture: Virtual Me

We do not come across great minds very often. It would be of great value for research to identify techniques to capture and analyze the decision making capability of
these minds. Virtual Me is focusing on achieving this. Virtual Me is a concept which focuses on capturing the patterns of a user based on numerous criteria such as reply pattern, browsing style, areas of interest, setting multiple filters and so on. This project focuses on a technique which could be a useful addition for Virtual ME’s feasibility. The project focuses on identifying reply patterns based on email conversation logs.
2. VIRTUAL COMMUNICATION ASSISTANCE

2.1 Scope

This project’s scope is limited to identifying the reply pattern. The project serves as a prototype for future enhancement and also as a foundation for research on Virtual Me. The project works on real time emails and identifies the reply patterns for these emails. In addition to identifying reply pattern the project also identifies date references in emails and schedules appointments. This application uses Natural Language Processing for its core functionality and the Android platform for implementation. The project identifies the closest reply pattern for the selected message based on conversation logs. The messages for which replies are generated are selected based on a few criteria and assumptions. One of the important criteria is mail server selection. The application supports only three mail servers namely Gmail, Yahoo and Outlook. The modular block diagram in Figure 2.1 shows the important components of the system.

![Modular block diagram of the entire system](image)

Figure 2.1: Modular block diagram of the entire system
Upon retrieving the email message, it is cleaned to remove all the ambiguous characters. After removing unwanted characters the message is tagged with identifiers. The cleaned message is parsed using the Stanford Parser which assigns parts of speech (POS) to each word in the message. The message is further parsed to sort it according to the POS to build a reply pattern. Also the raw message is passed to the Android device to identify date and time and add events accordingly. Once the Android device receives the message it identifies the date from the message. The date is identified only when the message has some kind of date mentioned in it. The date mentioned can be in any form ranging from straight forward representation such as “Monday 3 PM” to “fortnight from now”. The date is then used for creating calendar events. Finally a graph is generated, which shows the closest pattern or structure the reply might be in.
3. REQUIREMENTS

The application needs to satisfy few requirements.

3.1 Plain English Description

1. The application must be able to connect to the mail server.
2. The application must be able to receive email.
3. The application must be able to detect unwanted emails.
4. The application must read the unread message.
5. The application must be able to extract the entire message thread.
6. The extracted message thread must contain all the previous messages and its respective replies.
7. The application must be able to identify the sender’s mail server and must clean the message thread accordingly.
8. The application must be able to parse the cleaned email and tag each word with its respective parts of speech.
9. The parsed email must be further parsed to generate a new dataset based on parts of speech.
10. The application must be able to parse the message and identify date and time, if the message contains it.
11. The application must create and/or reschedule calendar appointments based on the date and time parsed.
12. The appointment created must be based on the sender’s priority.
13. The application must update the sender informing about the appointment.
14. The application must be able to generate and display graph based on the dataset.
3.2 Functional Requirement

These are the important functionality for the system to perform.

1. First, the system must be able to fetch emails without user intervention and verify it.

2. Fetching the email: First and the most required step for the application is fetching the emails from the mail server. The entire project is built on processing email messages, so it is very important for this functionality to execute smoothly.

3. Verifying the email: Every mail received is not a meaningful mail. Everybody end up receiving spam mails, junk mails, advertisement mails and so on. All such messages cannot be processed. Messages to be processed have to meet with few criteria and this functionality ensures it.

4. Verifying sender information: Every message has to meet certain criteria to be treated as a valid message for generating output. Verifying sender name and email address is the easiest way of filtering the messages.

5. Second, the system must be able to parse the email message generate the dataset.

6. Parsing email message: Once a valid message has been obtained, it needs to be tokenized.

7. POS tagging: The tokens must be assigned with their respective parts of speech, such that each token can be identified by their parts of speech (POS).

8. Dataset generation: New data must be generated based on the parts of speech assigned to each token.
9. Third, the system must be able to create and reschedule calendar appointments automatically.

10. Date identification: The system must be able to identify date patterns from the given message.

11. Updating calendar: The system must be able to automatically update the calendar events based on the date and priority of the sender.

12. Fourth and finally the system must be able to display notification and results.
4. DESIGN

This application follows client server model. Section 4.1 and 4.2 describe the various components involved in the application.

4.1 Client Design

This section describes the components involved in the client side of the application.

4.1.1 Reply Pattern

Once the dataset is generated, the raw data present in it has to be represented in some kind of an output. This pattern is responsible for computing the reply data based on probability theory. The reply pattern data is represented by plotting a graph.

4.1.2 Calendar Updater

The calendar updater component is responsible for the following operations.

4.1.2.1 Date recognition

Once the message is received and cleaned on the server, the message has to be checked for date time strings. Date time strings can be in any pattern ranging from, straight forward representation like ‘31\textsuperscript{st} July at 1 PM’ to complex representations like ‘a fortnight later at noon’. This component used date parser to identify the date string.

4.1.2.2 Event Verification

Once the date string is identified it is used by calendar class to check for appointments. If am existing appointment event is found then the priority of the sender is compared with the priority of the person to whom the appointment was initially assigned. If the priority of the sender is higher, then the sender is provide the slot and the other person is assigned a new slot.
4.2.2.3 User Approval:

Due to the security concern of adding event directly to the calendar, the application requests the user to approve the rescheduling. By doing so the user is alerted about the event. Also under many situations, personal choices and priority may vary from choices on a database. Therefore upon user approval the rescheduled event is added at the newly computed date and time and the new event is added to the requested time.

4.1.3 Notification Generator

The notification component is responsible for sending automated acknowledgement messages to the sender and other people (if rescheduled) about the appointment. Also this component is responsible for creating toast.

4.2 Server Design

The server consists of 3 components. The three components are email fetcher, POS tagger and dataset generator. The first component email fetcher is implemented using JavaMail API.

4.2.1 JavaMail API

JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications. JavaMail API is used due to the security concerns and inability of Android to provide access to email client and other email application. JavaMail supports both IMAP and POP3 protocols for accessing emails.

4.2.2 Stanford Parser

The second component POS tagger is implemented using Stanford Parser API. Stanford Parser is a natural language parser which takes in any natural language string and parses each token into its respective parts of speech. When the natural string is
tokenized, the tokens are verified (based on the context) against Stanford defined
grammar. Upon verification the token is tagged with its respective parts of speech such as
noun, verb, adjective, conjunction and so on. Table 4.2.2 describes various tags and its
description. Please note, the table does not contain all the tags generated by the parser.
Only the tags mentioned in the table are used for the project.

<table>
<thead>
<tr>
<th>POS Tag</th>
<th>What it Denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Conjunction</td>
</tr>
<tr>
<td>CD</td>
<td>Cardinal Number</td>
</tr>
<tr>
<td>DT</td>
<td>Determiner</td>
</tr>
<tr>
<td>IN</td>
<td>Preposition</td>
</tr>
<tr>
<td>JJ</td>
<td>Adjective</td>
</tr>
<tr>
<td>MD</td>
<td>Modal</td>
</tr>
<tr>
<td>NN</td>
<td>Noun</td>
</tr>
<tr>
<td>POS</td>
<td>Possessive ending</td>
</tr>
<tr>
<td>PRP</td>
<td>Pronoun</td>
</tr>
<tr>
<td>RB</td>
<td>Adverb</td>
</tr>
<tr>
<td>TO</td>
<td>To</td>
</tr>
<tr>
<td>VB</td>
<td>Verb</td>
</tr>
<tr>
<td>WP</td>
<td>Wh-pronoun</td>
</tr>
</tbody>
</table>

Table 4.2.2: Parts of speech and its description

4.2.3 Dataset Generator

The third and final component on the server side is the dataset generator. The dataset
generator is a java class which further parses the parsed output from the Stanford parser.
This component is responsible for separating the tokens based on the POS and building a table based on the respective POS count. The information collected in the table is used in later stages for developing reply pattern graphs.

4.3 Use Case Diagram

Level 1: Main system

![Use Case Diagram](image)

Figure 4.3.1: Level 1: Main use case diagram
Level 2: Fetch email

Figure 4.3.2: Level 2 use case for Fetch email

Level 2: Manage Calendar

Figure 4.3.3: Level 2 use case for Manage calendar
4.4 Class Diagram

Figure 4.4.1: Server Class Diagram
Figure 4.4.2: Client Class Diagram
4.5 Activity Diagram

Figure 4.5.1: Server activity diagram
Figure 4.5.2: Client activity diagram
4.6 Sequence Diagram

Figure 4.6.1: Sequence diagram for Fetch email use case
Figure 4.6.2: Sequence diagram for managing calendar use case
4.7 User Interface

The application is designed considering metro UI in mind. The UI looks very similar to the metro UI, which is used in Windows 8. The application has four basic views on the client side and just one view on the server side.

Server View: The server’s user interface is developed using Swing GUI from Java foundation Classes. The UI consists of a button, text area and box layout. The button acts as the only single point of interaction between the user and the server.

Client Start up View: The client start up UI consists of a list view and two buttons as shown in Figure 4.7.1. The list view displays the connection and progress details of the application, when executed. The ‘Start Server’ button triggers the server to start its process of fetching emails, cleaning the email and so on. The ‘Contacts’ button opens the contact view.

Contact View: This view lets the user manage contact database. The manage operation that are provided include insertion of new contact, deletion of existing contact, updating existing contacts and so on. The Contact View also lets the user access the contact list view through a button press.

Contact List View: This view lets the user view the contact information present in the contact database.

Graph View: This view is created by the external library AChartEngine. This particular view lets the user see the result in a graphical form. The library provides support to many graphs such as line graph, bar graph, pie chart and so on.
Figure 4.7.1: Client Startup View

Figure 4.7.2: Contact View

Figure 4.7.3 Contact List View
5. IMPLEMENTATION

The application follows client server architecture. The server is implemented in java and the client is implemented in Android using Eclipse IDE, SDK version 4.2. The client and the server communicate through TCP/IP protocol.

5.1 Assumptions:

The following assumptions are made for the implementation,

- The application receives emails only from Gmail, Yahoo or Outlook server.
- The new message received is part of an existing thread with at least six to seven previous conversations.
- The message received is from one of the contacts present in the custom contact database.
- There cannot be two contacts with same priority. Every contact has its own priority.
- The application does not handle errors. The application follows the happy path.

5.2 Client Side:

The mobile client is developed using Android SDK integrated into Eclipse IDE in Windows 7. The mobile client supports a minimum SDK version of 8 (Android 2.2 Froyo) and a maximum SDK version of 17 (Android 4.2 Jelly Bean). The application requires access to Content Providers. Content Providers are responsible for managing accesses to the data present in the Android device. Content Providers are responsible for defining the mechanisms for maintaining data security. In addition Content Providers provide interface to connect the data available in the one process/application to code running in another process.
The client is initially designed to use two Content Providers namely, Contact Provider and Calendar Provider. The Contact Provider enables the application to access the data available in the contact database. The contact provider lets the application access all the contacts present in the device. Similar to Contact Provider, Calendar Provider allows access to the all the calendars and its events which are linked to the device. But, as mentioned in the limitations, the Contact Provider was not of much use and a new database to maintain the sender’s priority is created.

The application on the client side is designed with a simple user interface to provide easy usability. As discussed in the previous section, the client does the major operations such as, scheduling and rescheduling the appointments, send acknowledgement and predict the result pattern.

Once the server process is completed the client reads the dataset file from the server. The client connects to the server, downloads and reads two files. The file where the cleaned message is written is downloaded and read. Next the file to which the dataset is written is downloaded and read. On reading the message the application stores the entire message in a string array. The string is then passed to the date class.

Figure 5.2.1: Date computed by the Date Parser
The date class parses the message and identifies the date and time from the message as shown in Figure 5.2.1. Once the date and time is obtained, the Calendar Provider is invoked to check for existing event. If an existing event is found, then the calendar class queries the event and gets the name of the person for whom the event is created. The application queries the priority database to find the priority of the sender and the existing person. Based on the query results, if the sender holds higher priority, then the existing event is rescheduled to the next available slot. The reschedule is not approved until the user approves it.

Since it is very difficult and sometimes impossible to define what the user may think, the application does not automatically add or reschedule the event to the calendar. The application reschedules the calendar based on the user’s priority, but does not add the event until the user approves it. Once the user approves the rescheduling, the existing event is added to a new time slot and the new request is added as a new event to the requested date and time. The application uses implicit intents to add the event, so that Android operating system can decide the most efficient way of performing the task requested (Code Snippet 5.2.3). Figure 5.2.2 shows the invoking of an implicit intent.

Upon rescheduling the calendar, the application sends the sender and the rescheduled event person an acknowledgement about the appointment and the rescheduling that happened.

5.3 Server Side:

Since Android does not provide any API to read emails from the default email client or other email applications, it is very difficult to access emails in the client application. To solve this issue a web server is implemented using Apache Tomcat. The
server and the client communicate through TCP/IP. Once the server starts, it keeps
listening for the trigger command form the client. This application uses
“VCATAMUCC” as its trigger command and communicates through port 4444 (Code
Snippet 5.3.1).

![Figure 5.2.2: Implicit intent requesting user approval to add an event](image)

Once the server is triggered, it calls the email class which in turn fetches the
e-mails from the mail server. The class uses IMAP protocol to communicate with the
email server (Code Snippet 5.3.2). For simplicity, the application is uses the Google mail
server. The Google account credentials are hard coded in the server for easy
implementation. Since the message fetched is in thread form, it requires cleaning before
being sent to the parser. Therefore, the message is sent to the cleaner class to have it
cleaned so it is easier to parse and sort the messages and its reply. Figure 5.3.1 and Figure 5.3.2 shows the message before and after it has been cleaned. Note, when the cleaning happens, the message and its reply are tagged with keywords “QUES” and “REPL” to provide with easy identification for the later stages.

Figure 5.3.1: Message imported from the server before cleaning

<table>
<thead>
<tr>
<th>QUES</th>
<th>Do you free to watch a show on 3rd August at 10am and go for swimming break</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPL</td>
<td>I will. Let me know before you book the tickets for the show Perla.</td>
</tr>
<tr>
<td>QUES</td>
<td>Hey, would you be interested in going to an Italian restaurant?</td>
</tr>
<tr>
<td>REPL</td>
<td>I would like that too. Please go ahead.</td>
</tr>
<tr>
<td>QUES</td>
<td>Dirty, then sounds great. Can we finish and go watch a movie and then go swimming break</td>
</tr>
<tr>
<td>REPL</td>
<td>No, I am planning to stay too. If you are interested we can go visit together Perla.</td>
</tr>
<tr>
<td>QUES</td>
<td>Hey, I am staying back and planning to visit some places. What about you? Are you leaving break</td>
</tr>
<tr>
<td>REPL</td>
<td>Anyway, what is your plan after break? Are you staying in corpus or going out Perla.</td>
</tr>
<tr>
<td>QUES</td>
<td>You are welcome Perla</td>
</tr>
<tr>
<td>REPL</td>
<td>This is awesome. I will be there for your defense for sure Perla</td>
</tr>
<tr>
<td>QUES</td>
<td>I am defending my thesis on 1 August in the morning around 10 AM Perla</td>
</tr>
<tr>
<td>REPL</td>
<td>Do you have any questions Perla</td>
</tr>
<tr>
<td>QUES</td>
<td>No problem. I will be glad to attend your defense Perla</td>
</tr>
<tr>
<td>REPL</td>
<td>Hey thanks. I am actually coming for my defense. I am presenting my project on 10th July at 10am. Will you be able to attend my defense Perla</td>
</tr>
<tr>
<td>QUES</td>
<td>Good to hear from you. Thanks for asking about my project. It is going well. I am here on 10th July and let me meet that break.</td>
</tr>
<tr>
<td>REPL</td>
<td>Hello Perla. How are you. Hope your thesis is getting finished. I am planning to visit Corpus Christi on 10th July. Are you free that day Perla</td>
</tr>
<tr>
<td>QUES</td>
<td>Hello Perla. How are you. It is going Perla</td>
</tr>
</tbody>
</table>

Figure 5.3.2: Cleaned message tagged with identifiers
The cleaned message is sent to the Stanford Parser class to assign each token in the message with its respective parts of speech. Due to the size limitation, the process of sending the message to the Stanford Parser is done line by line in a loop until all the lines are processed and each token in the line is tagged with its parts of speech. Figure 5.3.3 shows the cleaned message assigned with its respective parts of speech (Code Snippet 5.3.3).

The parsed and tagged message is passed to the dataset generator class. The dataset generator class reads in sentence by sentence, where each sentence is categorized according to its respective part of speech. The categorized sentence is loaded in a CSV file. This CSV file is later accessed by the client to generate the graph based on the reply pattern. Table 5.3.4, shows the final generated POS frequency table.

![Figure 5.3.3: Tokens in the message tagged with their respective POS](image-url)

```plaintext
1 [AMES/MNP, Are/VBP, you/PRP, free/OJ, to/TO, watch/VB, a/DT, show/VN, on/IN, 3rd/OJ, August/NNP, at/IN
2 [REFL/VB, I/PRP, will/MD, let/TO, me/PRP, know/VB, before/IN, you/PRP, book/VB, the/DT, tickets/MNP, :
3 [AMES/VLB, Hey/MNP, ,/, when/NRB, you/PRP, are/VBP, coming/VBG, could/MD, you/PRP, bring/VB, some/DT,
4 [AMES/VLB, I/PRP, would/MD, like/VB, that/DT, too/TO, too/TO, Please/VB, go/TO, ahead/RF, Partha/NPN
5 [AMES/VLB, Sure/MNP, ,/, that/WDT, sounds/VBD, great/OJ, We/PRP, can/MD, first/NN, go/TO, watch/VB, a
6 [AMES/VLB, No/UB, ,/, I/PRP, am/VBP, planning/VBD, to/TO, stay/VB, too/TO, If/IN, you/PRP, are/VBP, in
7 [AMES/VLB, Yes/UB, ,/, I/PRP, am/VBP, staying/VBD, back/NN, and/CC, planning/VBD, to/TO, visit/VB, son
8 [AMES/VLB, Anyway/RB, ,/, what/WRP, your/PRP$, plan/NN, after/IN, thesis/NN, are/VBP, you/PRP, staying/
9 [AMES/VLB, You/PRP, are/VBP, welcome/JJ, Partha/NPN
10 [AMES/VLB, That/DT, is/VBD, awesome/OJ, I/PRP, will/MD, be/TO, for/IN, your/PRP$, defense/MNP, 
11 [AMES/VLB, I/PRP, am/VBP, defending/VBG, my/PRP$, thesis/NN, on/IN, 1/CD, August/NNP, in/IN, the/DT, m
12 [AMES/VLB, So/UB, when/NRB, are/VBP, you/PRP, defending/VBG, your/PRP$, thesis/NN, Partha/NPN
13 [AMES/VLB, No/DT, problem/NN, I/PRP, will/MD, be/TO, glad/JJ, to/TO, attend/TO, your/PRP$, defense/MNP, 
14 [AMES/VLB, Hey/MNP, thats/VBD, good/OJ, I/PRP, am/VBP, actually/RR, coming/VBG, for/IN, my/PRP$, defen
15 [AMES/VLB, Good/OJ, to/TO, hear/VB, from/IN, you/PRP, Thanks/NNS, for/IN, asking/VBD, about/IN, my/FRP
16 [AMES/VNB, Hello/NPP, Burtak/VBD, ,/, How/NRB, are/VBP, you/PRP, Hope/VBN, your/PRP$, thesis/NN, is/NN
17 [AMES/VLB, Hello/NPP, Partha/NPP, ,/, How/WRP, is/NN, it/PRP$, going/VBG, Burtak/NPP
```
Once the dataset is generated, the POS tagged messages are filtered based on specific tokens. The filters for the tokens are tags such as adverb, verbs, interjection and so on which describe the action or the context of the message. The filtered message which requires a reply is then compared with the other filtered message to find similarities. Also the distance between the token based on the order in which they appear are also calculated. To calculate the similarity, the cosine similarity algorithm is used. According to [4], Cosine similarity algorithm treats each message as a vector and each token in it as a dimension. The algorithm uses these vectors and dimensions for producing meaningful similarity among them. In addition, Levenshtein distance algorithm is used to find the distance between the sequences of the tokens. By doing so, it is easy to find how similar two messages are based on the verbiage order. The Levenshtein distance algorithm helps in enhancing the cosine similarity comparison. Code snippet 5.3.4 shows the usage of Levenshtein algorithm with cosine similarity algorithm. For example, let us consider two scenarios with two lists of words.

Scenario 1: List 1{play eat swim watch}, List 2{watch eat swim play}

Scenario 2: List 1{play eat swim watch}, List 2{watch eat swim play}
Cosine similarity computed when comparing List 1 with List 2 in scenario 1 is 100%. Also the cosine similarity is the same in scenario 2. In this case, both the scenario gives the same cosine similarity value, making it difficult to decide the closest match. To overcome this problem, Levenshtein distance algorithm is used. The Levenshtein distance measures the distance between List 1 and List 2. By doing so it is found for scenario 1, the distance is 14 and for scenario 2 the distance is 6. Smaller distance indicates the order in which the words appear is preserved. Figure 5.3.5 shows how the Levenshtein distance is computed.

```
play eat swim watch
watch eat swim play
```

Distance computed is 14

```
play eat swim watch
eat play swim watch
```

Distance computed is 6

Figure 5.3.5: Distance computed for scenario 1 and scenario 2

5.4 Limitations

- One of the first limitations faced is the compatibility of Android operating system with JavaMail API and Stanford Parser. The Android device is not able to support both JavaMail API and Stanford Parser API. This limitation is handled by moving the APIs to a server.
- Next limitation faced is with the Stanford CoreNLP parser. The parser requires 64-bit system with a minimum of 3 GB of RAM space to create virtual machine. Since the server used is a 32-bit system with 4 GB RAM, Stanford CoreNLP Parser is not used. Instead another version of Stanford Parser is used. The new
version of Stanford parser is limited to just analyzing the natural string to its respectively parts of speech.

- Since the project focuses on creating a real time proof of concept, the application is processing just one message. The message being processed has to be a chain message with a minimum of six to seven previous replies. For the sake of convenience, only proper mails with previous replies are considered and all other mail is discarded.

- While testing the server implementation another defect was found. The parser does not parse correctly any string whose length is more than 250 characters. When a string of length 250 characters or more is parsed, each was token is just assigned with ‘X’ as its respective parts of speech. In order to overcome the problem, the design had to be slightly adjusted. The new adjustment is, now the parser does not take the entire message as a single input, instead the message is broken into smaller substrings and fed to the parser one by one.

- The final limitation faced is with the contact database. The default contact database did not provide any option to set a priority for the people present in the contact list. Thus, a new priority database is created to maintain the sender’s information and their priority.
6. TESTING, RESULTS AND EVALUATION

6.1 Testing

The application is tested by distributing a beta version. The users are first explained about the application such as what it can do, what should not be done and how it has to be done. Next, the application is tested with multiple email servers. The application successfully processes the messages from the three approved mail servers (Gmail, Yahoo, and Outlook). The first defect faced while testing was that the application kept crashing when generating the graph. The reason for the crash is running a non UI process on a UI thread. The graph class fetches the real time data from the web server. This process of fetching data has been executing on the UI thread which led to the application to crash. Upon fixing the issue the graph is generated with out any error.

The next phase, involved testing the application for rescheduling. Multiple emails requesting appointments are parsed. The dates object obtained from the emails are used for testing the appointment scheduler module. The application did not crash or take a long time to reschedule the event. Events are rescheduled successfully based on the priority. Figures 6.2.4 and 6.2.5 show the event creation when the calendar is free. Figures 6.2.6 and 6.2.7 show event rescheduling. As mentioned earlier, event rescheduling is based on the contact priority. Figure 6.2.3 shows the contact’s priority.
6.2 Results

Figure 5.3.1, 5.3.2, 5.3.3 and Table 5.3.4 show how the message is handled in each step to derive the dataset. Figure 6.2.1 show the cosine similarity computed using Levenshtein distance algorithm.

| Are watch go swimming                      |
| are coming bring 0.0                       |
| sounds first go watch then go 0.4472136    |
| am staying back planning visit about Are 0.18898225 |
| are 0.0                                   |
| am defending 0.0                          |
| be attend 0.0                             |
| hear asking is going am here lets meet then 0.0 |
| going 0.0                                 |
| 1                                         |

Figure 6.2.1: Cosine Similarity between the new message and the previous messages

The last line “1” seen in Figure 6.2.1, points a position in the dataset. I.e. The position points to a previous message in the dataset. This previous message has the highest similarity to the new message. Based on this result, the application infers the reply for the new message may be similar to the previous message’s reply. The data is fetched from the dataset and drawn as graph as shown in Figure 6.2.2.

Figure 6.2.3, 6.2.4, 6.2.5, 6.2.6 and 6.2.7 show the appointments created and rescheduled.
Figure 6.2.2: Reply pattern plotted as a graph along with its question

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Email</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 victor</td>
<td>oldman</td>
<td><a href="mailto:vikioldman@gmail.com">vikioldman@gmail.com</a></td>
<td>4</td>
</tr>
<tr>
<td>7 John</td>
<td>Snow</td>
<td><a href="mailto:Johnshnow@outlook.com">Johnshnow@outlook.com</a></td>
<td>1</td>
</tr>
<tr>
<td>9 burak</td>
<td>ersoy</td>
<td><a href="mailto:burakersoy1903@gmail.com">burakersoy1903@gmail.com</a></td>
<td>2</td>
</tr>
<tr>
<td>15 kashif</td>
<td>ansari</td>
<td><a href="mailto:kashif_ansari50@yahoo.com">kashif_ansari50@yahoo.com</a></td>
<td>5</td>
</tr>
<tr>
<td>16 partha</td>
<td>krishna</td>
<td><a href="mailto:Parthak@outlook.com">Parthak@outlook.com</a></td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 6.2.3: Custom contact list with priority
Figure 6.2.4: calendar with no events

Figure 6.2.5: Event created at the requested time

Figure 6.2.6: Calendar with event

Figure 6.2.7: Event rescheduled based on the priority
6.3 Evaluation

The results are evaluated based on the user input. The user comes up with couple of possible replies for the message. This message is then parsed and its respective POS is identified and added to the dataset. Once the POS frequency is added to the dataset then the frequency is compared with the reply identified in the result section. Figure 6.3.1 shows the two possible sample messages considered for the evaluation. Figure 6.3.2 shows the comparison among the messages based on the POS frequency.

“Yes I am free, let’s do it. Partha”

“Sure no problem, I will join you. Partha”

Figure 6.3.1: Assumed possible replies

Figure 6.3.2: POS frequency comparison
7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

The application has taken a very tiny portion of Virtual Me and worked towards identifying the reply pattern based on previous email logs. The reply pattern is identified by finding an existing message which has the highest similarity with the new message and using its reply for the pattern. In addition the project has also identified the issues and the limiting factors which pose huge challenge for further enhancements.

7.2 Future Work

This project provides proof of concept for a very minute portion of Virtual Me. This project is intended to act as a starting point for further researches on virtual cloning. Future enhancement for this project can be approached in two ways, thesis approach and project approach.

In project approach, the application can be improved to handle multiple emails and also single email involving multiple participants. Handling email which involves multiple participants can be very tricky since the entire email conversation is observed by the other participants. In such conditions, the user may or may not be replying in the same way they do on a one on one email. An approach to handle such special situations could be, identifying the individual reply data set between the user and each of the participants strictly based on the previous one on one emails. Also another data set is identified among the user and the other participants based on the multi-participant email.

Upon obtaining two reply data sets, the replies from one data set can be compared with the replies in the other data set. By doing so, similarity among the messages could
be found. This similarity could act as the bridge to connect or establish relation between the one on one message and multi-participant messages, to produce the reply pattern. The application can further be enhanced by comparing the emails based on context similarity. Context similarity can produce better pattern identification. By doing so the difference between strings “I eat no meat” and “No, I eat meat” can be easily found.

Another enhancement for the project approach could be the ability of the application to identify legitimate and valid email. Spam emails, subscription emails and so on can be filtered.

Also the calendar rescheduling can be improved. The improvement can include, assigning same priority to multiple users and handling rescheduling among them. The rescheduling can further improved to decide the priority of the event based on the purpose of the event. A doctor's appointment could be more important than meeting a person with priority one. The application could be improved to handle such solution.

As for the thesis approach, Stanford Parser can be substituted with custom parser. A new parser can be created which performs only the required operations. By doing so, the time taken to process and parse the emails could be reduced to a considerable extent. Also the similarity among the messages could be exploited to build a heap, list, stack or any kind of data structure to map or hold similar and related message. In addition the mapping could be further extended to connect related words. This connection of words can form a tiny part of huge neural network of related words and its respective synonyms.

Another improvement could involve the use of more sources. Messages exchanged through SMS, chats, instant messages, blogs can all act as a good source. These sources
could be used to identify the users general reply style. The thesis approach can focus more on improving the algorithm to process the emails and develop the reply pattern.
8. BIBLIOGRAPHY


[8] https://java.net/projects/javamail/pages/Home


9. APPENDIX A: TEXTUAL DESCRIPTION FOR USE CASES

1) Title: Fetch email

Summary: This use case allows the user to import emails from the mail servers to the local server, which can be then used for processing.

Actors: JavaMail API and Application user.

Pre-condition:

- Mail server running
- Local Server running
- User has valid credentials to access the mail server

Flow of events:

Main successful scenario:

- User provide valid credentials
- Local server able to connect to mail server
- Local server able to import the emails to the local server

Alternate scenario

- User provide wrong credentials
- User not registered to the mail server
- Local server unable to establish connection to the mail server

Post-condition:

Same as precondition

2) Title: Parse email

Summary: This use case allows the user to parse emails and assign each token in the mail with its respective parts of speech.
Actors: Stanford Parser API and Application user.

Pre-condition:

- The message is cleaned
- The message is of a valid size

Flow of events:

Main successful scenario:

- The cleaned message is passed to the parser sentence by sentence.
- The message is tokenized
- Each token is assigned with its respective parts of speech

Alternate scenario

- The message is not cleaned
- The message is not broken into sentences
- Parser gets overloaded
- Error is displayed

Post-condition:

Same as precondition

3) **Title: Manage calendar**

Summary: This use case helps the user to automatically create calendar entries and reschedule existing calendar entries.

Actors: Application user, database

Pre-condition:

- Message is accessed from the server by the client
- Message contains date and time in some form of natural string
Calendar Provider access enabled

Flow of events:

- Main successful scenario:
  - The message is parsed
  - Date and time is identified from the natural string
  - Calendar is queried for the date
  - Event is created
  - Existing appointment is rescheduled

- Alternate scenario:
  - Message does not have any date and time
  - Calendar Provider access not available
  - Event not created

Post-condition:

Same as precondition

4) **Title: Send acknowledgement**

Summary: This use case allows the user to send acknowledgement to the sender.

Actors: Application user, JavaMail API

Pre-condition:

- Calendar event is created
- The sender information is available

Flow of events:

- Main successful scenario:
  - The sender information is auto filled in the message template
The message is sent to the sender

Alternate scenario

Calendar event not created

Message not sent

Post-condition:

Same as precondition

5) **Title: Display result**

Summary: This use case allows users to view the model graph developed using the data from the dataset.

Actors: Application user

Pre-condition:

- The data is accessed from the dataset
- The result pattern is computed based on the obtained data

Flow of events:

- Main successful scenario
  - The data is obtained
  - The data pattern is computed
  - The data is passed to the graph API
  - The graph is drawn successfully

- Alternate scenario
  - Unable to compute the reply pattern
  - Graph not drawn
  - Error message displayed
Post-condition:

Same as precondition

6) Title: Clean message

Summary: This use case lets the user clean the extracted message conversation.

Actors: Application user

Pre-condition:

- The message is extracted from one of the accepted mail server
- The extracted mail is part of an existing conversation

Flow of events:

- Main successful scenario:
  - The message is matched to the respective mail server
  - The respective cleaner class is invoked and the message is cleaned
  - The cleaned message is tagged with identification tags
  - The message is written to a file

- Alternate scenario:
  - The message does not belong to any accepted mail servers
  - Error message is displayed

Post-condition:

Same as precondition
10. APPENDIX B: CODE SNIPPETS

Code Snippet 5.1.1: Add event to calendar

```java
Calendar beginTime = Calendar.getInstance();
beginTime.setTime(date);
beginTime.add(Calendar.HOUR, extraTime);
Calendar endTime = Calendar.getInstance();
endTime.setTime(date);
endTime.add(Calendar.HOUR, extraTime + 1);
Intent intent = new Intent(Intent.ACTION_INSERT)
    .setAction(Events.CONTENT_URI)
    .putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,
        beginTime.getTimeInMillis())
    .putExtra(CalendarContract.EXTRA_EVENT_END_TIME,
        endTime.getTimeInMillis())
    .putExtra(Events.TITLE, title)
    .putExtra(Events.DESCRIPTION, description)
    .putExtra(Events.AVAILABILITY, Events.AVAILABILITY_BUSY);
startActivity(intent);
```

Code Snippet 5.1.2: Fetch data from custom database

```java
public String getData() {
    // TODO Auto-generated method stub
    String[] columns = new String[] { KEY_ROWID, KEY_FIRSTNAME,
        KEY_LASTNAME, KEY_EMAIL, KEY_PRIORITY };
    Cursor c = ourDatabase.query(DATABASE_TABLE, columns, null, null, null, null, null);
    String result = "";
    int iRow = c.getColumnIndex(KEY_ROWID);
    int iFirstName = c.getColumnIndex(KEY_FIRSTNAME);
    int iLastName = c.getColumnIndex(KEY_LASTNAME);
    int iEmail = c.getColumnIndex(KEY_EMAIL);
    int iPriority = c.getColumnIndex(KEY_PRIORITY);
    for (c.moveToFirst(); !c.isAfterLast(); c.moveToNext()) {
        result = result + c.getString(iRow) + " " + c.getString(iFirstName)
            + " " + c.getString(iLastName) + " " + c.getString(iEmail)
            + " " + c.getString(iPriority) + "\n";
    }
    return result;
}
```
// Our first data
TimeSeries series = new TimeSeries("Message");
for (int i = 0; i < x1.length; i++) {
    series.add(x1[i], y1[i]);
}

// Our second data
TimeSeries series2 = new TimeSeries("Reply");
for (int i = 0; i < x2.length; i++) {
    series2.add(x2[i], y2[i]);
}

XYMultipleSeriesDataset dataset = new XYMultipleSeriesDataset();
dataset.addSeries(series);
dataset.addSeries(series2);

XYMultipleSeriesRenderer mRenderer = new XYMultipleSeriesRenderer();
XYSeriesRenderer render1 = new XYSeriesRenderer();
XYSeriesRenderer render2 = new XYSeriesRenderer();
mRenderer.addSeries(render1);
mRenderer.addSeries(render2);
mRenderer.setCTitle("Parts of Speech");
mRenderer.setYTitle("Count");
mRenderer.setAxesColor(Color.GREEN);
mRenderer.setLabelsColor(Color.YELLOW);

Code Snippet 5.1.3: Generating graph

@override
public void actionPerformed(ActionEvent e) {
    // disable the start button
    startServer.setEnabled(false);
    mServer = new TCPServer(new TCPServer.OnMessageReceived() {
        @Override
        // this method declared in the interface from TCPServer
        // class is implemented here
        // this method is actually a callback method, because it
        // will run every time when it will be called from
        // TCPServer class (at while)
        public void messageReceived(String message) {
            if (message.equals("VCATAMUCF")) {
                messagesArea
                    .append("\nReceived request from client to fetch email. Process initiated\n")
                try {
                    VirtualAssistant va = new VirtualAssistant();
                    result = va.startAssistant();
                    if (result != false) {
                        mServer.sendMessage("Server Process Is Completed");
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                    System.out.println(e);
                }
            }
        }
    });
    mServer.start();

Code Snippet 5.2.1: Server Connection
```java
try {
    Properties props = System.getProperties();
    props.setProperty("mail.store.protocol", "imaps");
    Session session = Session.getDefaultInstance(props, null);
    store = session.getStore("imaps");
    store.connect("imap.gmail.com", "username@gmail.com", "******");
    folder = (IMAPFolder) store.getFolder("inbox");
    if (!folder.isOpen())
        folder.open(Folder.READ_WRITE);
    Message[] messages = folder.getMessages();
    for (int i = messages.length - 1; i >= messages.length - 1; i--)
    {
        String newMessage = "";
        Message msg = messages[i];
        Multipart multipart = (Multipart) msg.getContent();
        for (int x = 0; x < (multipart.getCount()) / 2; x++)
        {
            BodyPart bodyPart = multipart.getBodyPart(x);
            newMessage = newMessage + "" + (bodyPart.getContent()).toString();
        }
    }
}
```

Code Snippet 5.2.2: Fetch email from mail server

```java
String sent2 = str;
Tokenizers? extends HasWord> toke = tlp.getTokenizerFactory().getTokenizer(new StringReader(sent2));
List<? extends HasWord> sentence2 = toke.tokenize();
List<List<? extends HasWord>> tmp = new ArrayList<List<? extends HasWord>>();
tmp.add(sentence2);
sentences = tmp;
for (List<? extends HasWord> sentence : sentences)
{
    Tree parse = lp.parse(sentence);
    GrammaticalStructure gs = getNewGrammaticalStructure(parse);
    List<TypedDependency> td1 = gs.typedDependenciesCCprocessed();
    String testin = null;
    testin = parse.taggedYield().toString();
    returnString= returnString+"" +testin;
}
```

Code Snippet 5.2.3: POS Tagging
public float getSimilarity(final String string1, final String string2) {
    final ArrayList<String> str1Tokens = tokenizer.tokenizeToArrayList(string1);
    final ArrayList<String> str2Tokens = tokenizer.tokenizeToArrayList(string2);

    final Set<String> allTokens = new HashSet<String>();
    allTokens.addAll(str1Tokens);
    final int termsInString1 = allTokens.size();
    final Set<String> secStrToke = new HashSet<String>();
    secStrToke.addAll(str2Tokens);
    final int termsInString2 = secStrToke.size();

    // now combine the sets
    allTokens.addAll(secStrToke);
    final int commonTerms = (termsInString1 + termsInString2) - allTokens.size();

    // return Cosine Similarity
    return (float) (commonTerms) / (float) (Math.pow((float) termsInString1, 0.5f) * Math.pow((float) termsInString2, 0.5f));
}

Code Snippet 5.2.4: Cosine similarity using Levenshtein algorithm