Videation Assistant for Blind and Cognitively-Impaired Users

By

Yafim Landa

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 2011, in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical Engineering and Computer Science

Abstract

Blind and cognitively-impaired people often experience difficulties in pursuing activities independently, and must secure assistance from others or suffer a loss of efficiency. Having more information about their environment would mitigate the challenges that these users must face on a day-to-day basis. Current systems devised to address this problem are either too specialized or don't solve the "last hundred yards" problems, such as finding the door of a business. This 6.UAP project is the first step towards fulfilling the vision described in a five-year CSAIL project proposal [1] of developing contextually aware machines that can provide aid to blind or cognitively-impaired people in a natural manner. In this joint project with Peter Iannucci, another 6.UAP student, I have built a vest that helps its user find objects or navigate the environment through the use of the Wizard-of-Oz prototyping technique, and that serves as the first realization of this vision. This extensible system may serve as the groundwork on which the later iterations of the project will stand, as well as a debugging platform. In addition, the system can be used to evaluate the usefulness of the larger-scale project by utilizing remote human assistants called 'wizards' to serve as the back-end, long before the other requisite technology is developed.

1 Problem Description

Independent activity and social participation are important for long-term health and wellness. Visually- and cognitively-impaired people face a tradeoff between independence and efficiency, which often leads them to seek out help from others to perform tasks that are considered easy for those who do not suffer from the impairment. Situational details are not readily available for blind or cognitively-impaired people. A person who is trying to move around the world, for example, must solve many problems in rapid succession while maintaining awareness of his or her surroundings. Simply walking outside, we must watch out for breaks in the sidewalk, low hanging branches, moving vehicles, and other hazards. We must be able to recognize our location by locating landmarks, familiar people, or street signs; In addition, we must recognize when we have reached our destination and plan how to approach it safely. Once there, if the destination is unfamiliar, we must be able to recognize key features such as a concierge desk at a hotel or the elevators in an apartment building. If we are meeting an acquaintance, we must be able to recognize the acquaintance's features.

Non-technical solutions like canes and guidance dogs do not solve most of these problems, and current assistive technology fails to adequately address these issues. Most often, it has fallen victim to the "fallacy of the successful first step," where the technology may work in the lab but fails to function in real-world situations.

2 Proposed Solution

A group of researchers — referred to as the co-PI's in this paper — in MIT's Computer Science and Artificial Intelligence Lab (CSAIL) have embarked on a five-year journey to

build technology to address the problems described above. This project, lead by Professors James Glass, Robert Miller, Nicholas Roy, Seth Teller, and Antonio Torralba [1], aims to create a vest that will enable a blind user to better navigate the world. The user will wear this system, and it will store and update a model of the user's environment. Through the use of its sensors and model, the system will be able to, for example, alert the user of any hazards nearby; remind the user where it last saw the user's possessions (or any other object); and help lead the user to said object. The user will be able to interact with the system naturally through voice commands, and the system will provide feedback through vibration, spoken text, and Braille.

3 Sub-problem Addressed in 6.UAP Project

Creating such a system requires tremendous effort and pulls from many subareas of computer science, including computer vision and user interface design. Following the principles of iterative design, it is useful to build an evaluation prototype to test the validity of the idea, and to get a grasp of the problem scope. This 6.UAP deals with the first stage of the project, which aims to evaluate the proposal and to lay down the foundation for further expansion on the project. The sub-problem addressed in the project, therefore, is how to get the project from a proposal to a working prototype. Working together with Peter Iannucci — another 6.UAP student — I was able to create a first prototype of the system. This prototype provides the interface through which the blind user can interact with the system, and a human-powered back-end through the use of a human 'wizard.' Using this system, we will be able to perform technical and user studies in order to gain more information about the challenges that lie ahead. Our

approach allows the project to progress past the first evaluation stage, after which more expensive, more complex, and more detailed versions may be built on top of the existing prototype using the knowledge gained from the evaluation.

We wanted to complete the proposal outlined in the 'Baseline Evaluation' section of the group proposal [1]. Keeping in mind that our primary purpose was to advance the project from the proposal to the prototype stage, we had two concrete goals in mind:

First, we wanted to examine how a person from our target population would interact with such a system. To address this question, we decided to implement the first version as a Wizard-of-Oz prototype, which is a wide (full-featured front-end) and deep (full back-end support through a human 'wizard') prototype. The Wizard-of-Oz prototype provides us with the ability to simulate the user interaction with the complete proposed system by ignoring any present technological challenges on the back-end. Our prototype collects all data that passes through the system so that the user's interaction could be recorded during user studies and replayed at a later time. Because it records all of the data that passes through it, this prototype can also conveniently serve as a debugging platform for the future iterations of the project. Acquiring such a corpus of data (sensor readings and user utterances) will also be useful in establishing a ground truth for training and testing the more advanced stages of the system.

Second, we would like to build a platform upon which future contributors would be able to expand. We designed our system so that it would be extensible, modular, and reusable. To that end, our system uses or is compatible with existing software in the Robotics, Vision, and Sensor Networks (RVSN) group in CSAIL. We have designed the system so

that its different components could be implemented on any device and in any implementation language.

4 Design

4.1 Overview

The prototype is designed to use two humans: the first is the blind person (the 'user') who interacts with the vest front-end, and the second is the person who simulates the back-end (the 'wizard'). A user interface was designed for each type of person. As of now, the user must wear the vest, carry a laptop, and hang the Braille display around his or her neck. The wizard uses a laptop that has two windows open: a window that displays video RGB data from the vest-mounted Kinect and an interactive console that facilitates a dialog between the wizard and the user through text and speech.

4.2 Hardware

Our vest needs a few critical components in order to provide the desired functionality. First, it must have input devices so that the user could enter commands. This includes an audio input device (a microphone) to issue voice commands, user press-able buttons, and a keyboard for debugging. It must have the ability to show the user's environment to the wizard through a video feed. To accomplish this, we mounted a camera that provides RGB and depth data to the system and an inertial measurement unit (IMU), along with a GPS. Second, the system must provide feedback through output devices. The system can speak and play alerts through on-board speakers, provide haptic feedback through a vibrator, and write text output to a Braille display. There are also two screens — the

Android and laptop screens — for displaying debug output. For this prototype, we filled the roles of these components using a stripped-down Microsoft Kinect, an Android HTC G1 phone, and a Dell Inspiron 640m laptop running Ubuntu 10.10 or an Apple MacBook Pro 3,1 running OS X 10.6.6.

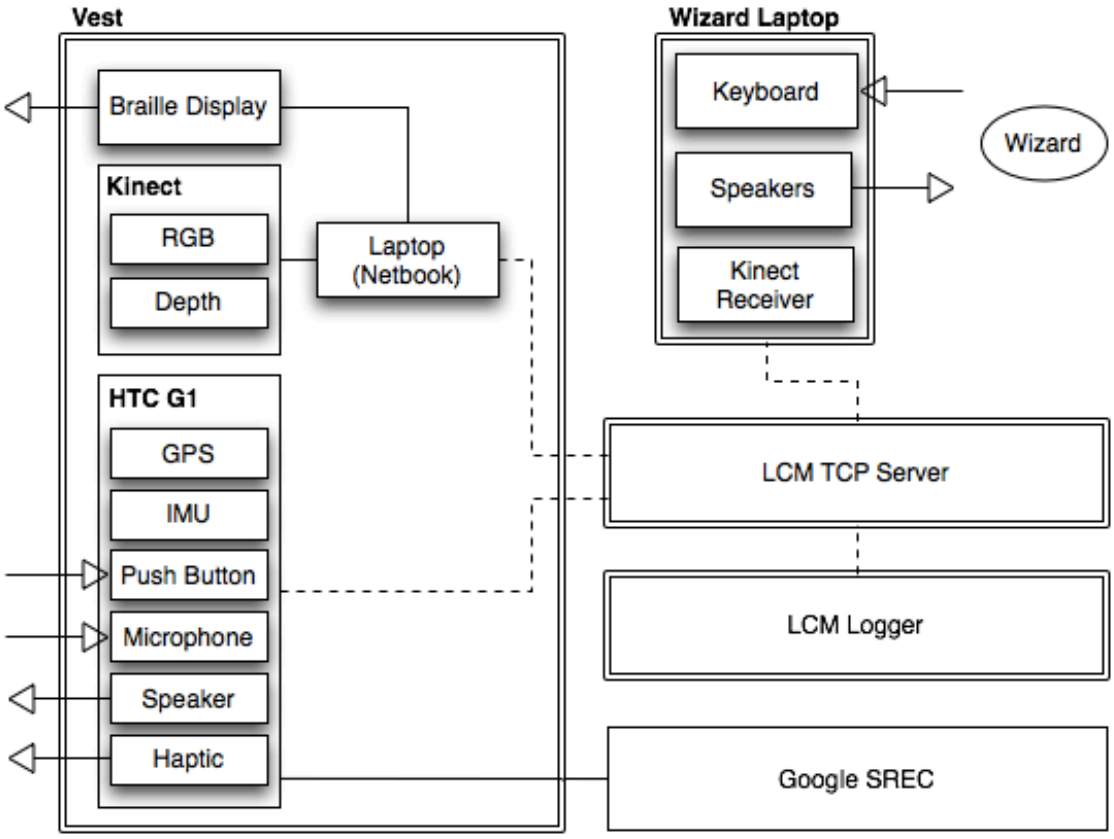

Figure 1. The hardware components: the wearable vest, the wizard laptop, the data logger, and the central server

The hardware roles are fulfilled as shown in Figure 1. The Android phone serves as the haptic and audio feedback device, and as the audio input device. The phone can also be used as the GPS and IMU, although we haven't implemented this functionality in this iteration. In addition, the phone has a variety of push buttons. The Kinect provides RGB and depth data to the laptop, which passes this information on to the rest of the system

through a centralized server using its network link. The laptop can issue Braille output to the Braille display, which itself has additional buttons (although these are typically used to scroll through long Braille messages). All of these components are mounted on a vest that can be seen in Figure ??. The physical vest was constructed by Jon Brookshire from the RVSN group, and already includes a Kinect device that has been stripped to the bare essentials. In addition, we mounted an Android G1 phone by putting it into a pocket on the vest. The positioning of the phone allows access to the microphone, the speaker, and a physical button. A SyncBraille device owned by the RVSN group can also be attached to the vest. We were considering using a belt clip to mount the Braille device, but it is currently free-hanging. Finally, the user must carry the laptop in his or her hands for this iteration of the prototype.

The wizard communicates with the user through the centralized server server over the network. The wizard has a laptop, which provides a keyboard to type messages to the user, a screen to see the dialog with the user, and a set of speakers that are able to output the user's recorded voice.

We have one dedicated server that channels all network traffic among the devices and one ma- chine (possibly the same machine as the server) that serves as a dedicated logger for all relevant data that passes over the network.

4.3 Software

To support our hardware design whereby any suitable device can serve any role and where all data is logged, we used LCM: a transport protocol for inter-process, inter-machine communication written in the RVSN group [2]. LCM relies on predefined

message types (LCM types), and broadcasts messages of each type over a specific LCM channel. We designed our messages to contain a destination field that specifies the desired output channel (for example, the haptic channel or the Braille channel). Now, any LCM device that is listening on the appropriate channel and has haptic or Braille capabilities can choose to process this message and provide feedback to the user. Using LCM not only allowed us to build on top of existing work in the RVSN group, but it also allowed us to decouple the abstract desired functionality from the concrete device that implements it. For example, our phone serves as a haptic device, an audio device, and a location device, but it can stop acting as a location device if we decide to add a dedicated GPS. To illustrate how our messages were structured, our LCM type specification files have been attached in Appendix A.1.

Since all of our inter-component communication transpires over LCM, we are able to feature logging and replay by recording LCM traffic and simulating it back, respectively. This functionality is already built into LCM, so we were able to take advantage of it with little effort. We had to consider some tradeoffs in our use of LCM. LCM relies on UDP multicast to broadcast its messages to all available listeners. However, we had difficulties getting LCM to work over UDP on our Android G1 phone, which was running the Android Cupcake operating system[1]. We instead decided to use a TCP LCM server to which all LCM devices would connect. While this provides the ability to easily log all data for replying, it is less optimal for certain types of data streaming, such as our Kinect RGB and depth data. TCP provides a delivery guarantee for our messages, a guarantee

[1] We used the older Cupcake operating system, API level 3, to ensure compatibility with the older G1 hardware.

that is usually good but can be an inhibitor for real-time audio and video data. We have provided all of our components with the ability to locate this centralized server before the component is started. On the other hand, LCM provides the ability for our components to easily interact regardless of what process or machine they are running on.

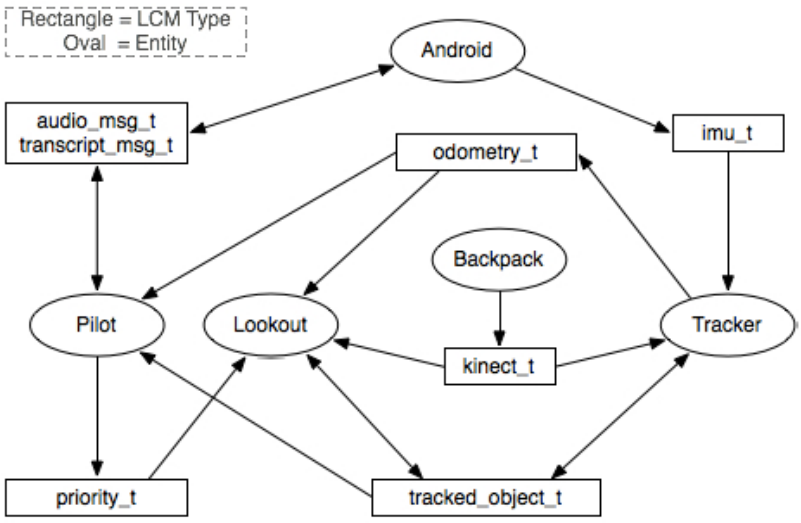

Figure 2. The software components of the Videation system.

This iteration of the project features only one wizard who decides what in the environment is important to the user and how to communicate that information to the user. However, these are conceptually two different tasks. We represented these two tasks as a Pilot component and a Lookout component, as can be seen in Figure 2. The Pilot is responsible for communicating with the user and establishing an ordered list of priorities. An example priority list might be (Hazards, Lab- mates, MIT Buildings, Street Signs). The Lookout can then find objects that fit into the priorities (in our example, these might be a low hanging branch, the street curb, the MIT Stata Center, and Peter on the other side of the street). The Lookout and Pilot interfaces will eventually be implemented by a set of algorithms, but are currently implemented with the single wizard application. In

fact, our system is designed in such a way that any component can be implemented in any language and on any device, or even with wizards (who essentially serve as 'human processors'). The next iteration of the project may feature two wizards – a Pilot and a Lookout – to evaluate this type of interaction, and our initial prototype was designed to make this extension easy to implement.

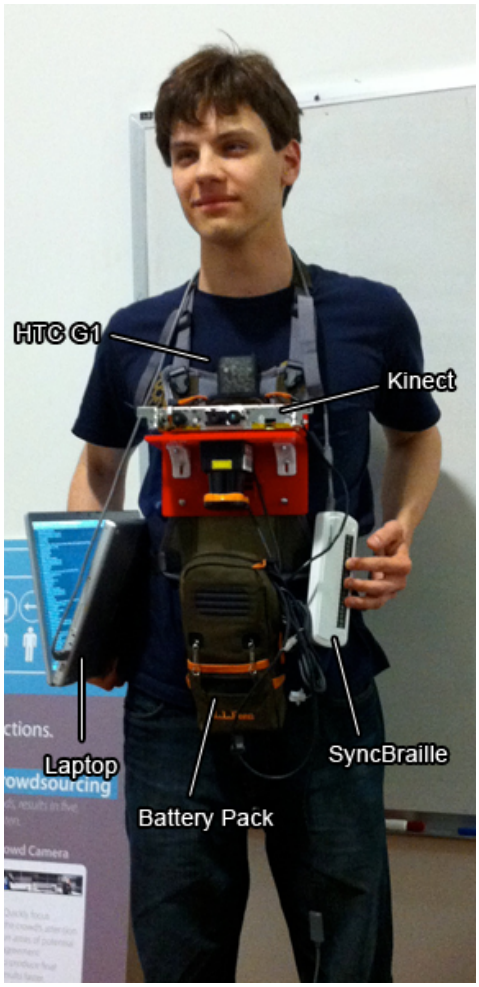

Figure 3. The vest that the user will wear, with labels for the various components.

5 Implementation

I mostly worked on the Android application, whereas Peter mostly worked on the wizard application. We both contributed to setting up the infrastructure, and our individual

contributions are described in that section. The initial vest was built by Jon Brookshire, and we added our own components to it. The vest can be seen in Figure 3.

5.1 Android Application

The Android phone has access to an accelerometer, GPS, a speaker, and a microphone, in addition to a screen (for debugging), a vibrator (for haptic feedback) and several buttons (although only one button was easily accessible from positioning of the phone on the vest). The Android operating system also has good support for text-to-speech (TTS), audio recording, and speech recognition (SREC). We were able to take advantage of these features to implement the features of the videation vest.

As mentioned above, the whole system used the LCM transport layer over TCP. We wrote a simple wrapper script that would publish the LCM TCP server IP address to a web-accessible location whenever a new server starts. Consequently, any videation LCM device is able to discover the LCM server by simply hardcoding the aforementioned web-accessible location into the LCM handler source code. The source code for this script can be found in Appendix A.2.

The phone listens to the transcript t message type, and responds differently depending on the contents of the LCM header. Regardless of the message, the phone outputs debug information to its built-in screen upon receipt of an LCM message. If the destination in a received message is set to DESTINATION_WORN_SPEAKER, then the phone speaks the message on the built-in speaker; if the destination is set to DESTINATION_BRAILLE_DISPLAY, then the phone alerts the user that a new Braille

message is available for reading by activating the vibrator. The transcript message header can be found in the second listing in Appendix A.1.

I decided early on to follow the Android convention of separating discrete tasks into individual Android Activity subclasses. An Android activity is a "single, focused thing that the user can do" [3]. Following this convention, the application launches into an LCMDiscoveryActivity. When a server is found, the VideationComm activity is launches, which handles user all further interaction. Upon a button press, an AudioRecordActivity is launched that captures audio from the on-board microphone, and the recording is stopped upon a second button press followed by starting the SpeechRecognitionActivity to convert the recorded audio into text using a Google API. The API provides a confidence level for each of its hypotheses, which is also transmitted over LCM. When the recording is stopped, the audio data is transmitted over LCM as Adaptive Multi-Rate (AMR NB) coded audio, sampled at 8kHz. AMR NB was used because it is particularly suitable for coding speech. It was suggested by Professor Teller to record the audio continuously into a circular buffer and run a speech detection module on the audio data. Although the speech detection code exists in the group's repository in 'ar-speech-command,' it has not yet been integrated into the Android application. It is unlikely that the existing code will be able to run in realtime on the Android phone, as the current code uses computationally intensive modulation frequency[2].

[2] Thanks to Jon Brookshire and Ekapol Chuangsuwanich for elaborating on the speech activity detection module.

Unfortunately, there is some delay in the audio recording module, and we require that the user wait for about 500ms to press the button before and after speaking. There is some haptic feedback from the vibrator that helps guide the user with the speaking timing, and there is haptic feedback to notify the user whether his or her speech was understandable. The user is prompted to speak with a 50ms and 60ms vibration separated by 50ms, and is acknowledged with either a 500ms 'error' vibration if the speech recognition confidence falls below a predefined threshold or a 50ms 'okay' vibration otherwise. The speech recognition confidence threshold can be specified in a centralized configuration file, along with the haptic feedback vibration patterns, the LCM lookup server ad- dress, the LCM channel name, and other such information. Please note that it appears that API level 11 (corresponding to the Honeycomb operating system release) allows for simultaneous on- device audio recording and speech recognition. In addition, the Android phone can act as an IMU and a GPS. While we can capture and transmit these values over LCM, we didn't have enough time to implement this functionality.

5.2 Braille

The Braille device is connected to the laptop. The laptop runs an application that listens for LCM messages that have destination set to DESTINATION_BRAILLE_DISPLAY, and outputs their content to the Braille display. Currently, the message is truncated to the length of the display line, which in our case was 20 characters.

5.3 Wizard Application

The wizard-facing part of the project consists of two components. The first component is the video display application, which is provided with the user's Kinect video data over

LCM. The second component is the interactive command-line interface through which a dialog between the user and the wizard is established. The video display application is a Python OpenGL wrapper for the kinect-glview application that was written in the RVSN group. This Python program functions like kinect-glview, but it uses a TCP LCM server instead of UDP multicast to look for video data over LCM. Peter also began implementing a version where the color data's saturation is decreased in areas where Kinect's depth data is unavailable, but this mode is erratic as of now. However, this code can serve as the base for somebody who wishes to investigate user interface ideas in the future.

The interactive command-line wizard application is run through rlwrap, which manages the read- line call and allows the user to type into the application while the rest of the application display is updating. After starting, it tries to connect to an LCM server by fetching its address from a hard-coded lookup server. If it fails to find an LCM server, then the program quits. However, if it is successful, it immediately presents a help message that instructs the user on how to use the program. The available options are to send a text message that will be spoken to the user, to send a message that will be written to the Braille display, and to play back the user's last recorded voice messages. The user is able to use shortcut keys for these different message types. In addition, the program reads in a list of canned messages from an external file. These canned messages can be sent quickly by using the number keys in case the wizard needs to react quickly.

The application listens to traffic over LCM, and displays any communication from the user. It detects voice (AMR) data over LCM, processes it using the ffmpeg utility, and

outputs it to the laptop's speakers. It also detects transcribed user utterances and outputs them to the screen for the wizard to see as well.

6 Infrastructure

We have had to set up some infrastructure to make the project possible, which is described in this section. Peter wrote the LCM server scripts and the wrappers for the existing utilities that enable them to use this functionality, and I wrote the initial LCM generation script, which Peter modified. The LCM generation script generates all of the LCM types for both Java and Python, and places them in the right directories within the Java and Python projects. This is done so that we wouldn't have to check any generated code into our version control system.

We have two web-accessible pages, a PHP script called 'up.php' and an HTML page called 'down.html'. The PHP script reads and writes the IP address of the LCM TCP server to the HTML page. Whenever we start an LCM TCP server, we send an HTTP GET request to 'up.php' with the server's IP address, which records it in the HTML file. Now, whenever 'down.html' is accessed, it provides the most recent LCM server IP address.

To make the LCM TCP server contact 'up.php' before starting, we wrapped lcm.lcm.TCPServer with a bash script. This script finds the machine's IP address by combing through the 'ifconfig' command and updates 'up.php' with this address. When the server quits, the IP address provided by 'down.php' is changed to '0.0.0.0' so that we know that no LCM server is available. The lcm-server script is provided in Appendix A.2 and 'up.php' is provided here:

Listing 1: up.php

```
<?php

$f = fopen('down.html', 'w');
fwrite($f, $GET['ip']);
fclose ( $f );

?>
```

To make use of our LCM TCP server, the existing utilities that are useful to us (such as the kinect- glview and lcm-logger utilities) are wrapped in scripts that first find the LCM server and then run the utility, following this general pattern:

Listing 2: Wrapper Script Pattern

```
IP="`curl http://lcm-discovery/down.html 2>/dev/null`"
if [ "$IP" = "0.0.0.0" ]
then
    echo "No LCM server found."
else
    # Run the utility with a pointer to the TCP LCM server
fi
```

7 Evaluation

Peter and I did an untethered test of the system. I put the vest on as shown in Figure 3 and walked to the eighth floor of the Gates tower of the MIT Stata Center. Peter was not paying attention to the wizard application, to simulate the system being off. After I got to the eighth floor, I moved to the intersection of building 36 and Stata, and asked Peter to help me get back to him through the use of the system. I tried not to rely on my vision, and used Peter's instructions, along with the Braille writing in the elevator. We managed to successfully navigate back to the RVSN group space on the third floor of Stata using the vest prototype in 383 seconds, capturing 405MB of data using the LCM logging utility. We were able to play this data back, and the logger reactivated every part of the

system, including the video stream and the verbal/textual dialog between Peter and me. We found that the Kinect stopped transmitting when I got too close to an obstacle and that the video stream dropped when I took the elevator down a few floors. However, these were not big issues, as the connections re-established themselves.

We also measured the approximate time delay that the system introduces into ordinary speech. We emulated a conversation by speaking back and forth to one another, with one of us repeating everything he said to simulate the system repeating what the user has said. We found that without the use of the system, we were able to communicate ten such messages in 21.2 seconds. Using the system at good network connectivity, we were able to communicate ten messages in 62.4 seconds.

8 Discussion

We feel that the system is at a point where further testing can be conducted. We were satisfied with our untethered test: we found that I was able to receive Peter's messages quickly and react appropriately to them, and that he could receive my data voice and video at a stable rate. In addition, when the network dropped, the system was able to reconnect and the demo would resume, at the expense of having to wait a few seconds for new instructions to appear.

The phone's UI worked, but it presented some problems when it queried Google's speech recognition API with a low WiFi signal. In the future, we may want to add more haptic feedback to indicate that the phone is busy, as I had to look at the phone's display to gauge its status. This only happened once during the course of the demo, however.

The next steps in the project would most likely entail testing the system against more users and collecting more technical data. There are still improvements that need to be made, such as building in speech detection into the phone and making the wizard application use a graphical user interface. In its current, command-line state, the wizard application does not support shortcut instructions very well. This was a problem in our untethered test, as Peter had a hard time trying to keep up with me as I executed his instructions.

Appendix A Code Listings

Appendix A.1 LCM Types

Transcript Type

```
package videation;

struct transcript_msg_t
{
    header_t header;
    int32_t  transcript_data_nbytes;
    byte     transcript_data[transcript_data_nbytes];
    double srec_confidence;
}
```

Transcript Header Type

```
package videation;

struct header_t
{
    // acquisition time (adjusted to host clock)
    int64_t  timestamp;

    int8_t source;
    int8_t destination;
    const int8_t SOURCE_WIZARD = 0;
    const int8_t SOURCE_USER_SPEECH = 1;
    const int8_t SOURCE_USER_TYPED = 2;

    const int8_t DESTINATION_WORN_SPEAKER = 0;
    const int8_t DESTINATION_BRAILLE_DISPLAY = 1;
    const int8_t DESTINATION_WIZARD = 2;
}
```

Audio Message Type

```
package videation;

struct audio_msg_t
{
    header_t header;
    int32_t  audio_data_nbytes;
    byte  audio_data[audio_data_nbytes];
}
```

Appendix A.2 LCM Discovery

```
#!/bin/bash

USE_FORCE=0
while getopts "f" Option
do
    case $Option in
        f ) USE_FORCE=1;;
        * ) exit 1;;
    esac
done

IFACE=`route get 18.0 | grep interface | awk '{print $2}'`
IP=`ifconfig $IFACE | grep inet[^6] | awk '{print $2}'`
OLD_IP="`curl http://lcm_discovery_addr/down.html 2>/dev/null`"

if [ "$OLD_IP" != "0.0.0.0" ]
then
    if [ "$USE_FORCE" -eq "0" ]
    then
        if [ "$IP" = "$OLD_IP" ]
        then
            echo "LCM server already running on localhost."
        else
            echo "LCM server already running at $OLD_IP."
        fi
        exit 0
    fi
fi

trap 'curl http://lcm_discovery_addr/up.php?ip=0.0.0.0; exit 0' 0
curl http://lcm_discovery_addr/up.php?ip=$IP
java -cp /usr/local/share/java/lcm.jar lcm.lcm.TCPService
```

Appendix B Software Setup Instructions

We have placed the software setup instructions in a collaborative Wiki so that anybody who runs into issues while installing our software could modify the instructions to document his or her experience. This Wiki can be accessed here:

http://groups.csail.mit.edu/rvsn/wiki/index.php?title=Videation_Project

References

[1] James Glass, Robert Miller, Nicholas Roy, Seth Teller, Antonio Torralba. Next-Generation Algorithms for Improved Context Awareness and Increased Independence.

[2] Albert S. Huang, Edwin Olson, David Moore. LCM: Lightweight Communications and Marshalling. Int. Conf. on Intelligent Robots and Systems (IROS), Taipei, Taiwan, October 2010.

[3] "Activity." Android Developers. Web. 20 May 2011.

<http://developer.android.com/reference/android/app/Activity.html>.