2018

Architectures for Real-Time Automatic Sign Language Recognition on Resource-Constrained Device

James M. Blair
University of North Florida, james.blair@outlook.com

ARCHITECTURES FOR REAL-TIME AUTOMATIC SIGN LANGUAGE RECOGNITION ON RESOURCE-CONSTRAINED DEVICES

by

James Blair

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computing and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

August, 2018

The thesis "Architectures for Real Time Automatic Sign Language Recognition on Resource Constrained Devices" submitted by James Blair in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee: Date

_____ _____

Dr. Ching-Hua Chuan

Thesis Advisor and Committee Chairperson

_____ _____

Dr. Roger Eggen

_____ _____

Dr. Sanjay Ahuja

Accepted for the School of Computing:

_____ _____

Dr. Sherif Elfayoumy
Director of the School of Computing

Accepted for the College of Computing, Engineering, and Construction:

_____ _____

Dr. William Klostermeyer
Interim Dean of the College

Accepted for the University:

_____ _____

Dr. John Kantner
Dean of the Graduate School

ACKNOWLEDGEMENT

I would first like to thank my advisor, Dr. Ching-Hua Chuan. Her mentorship and advice provided invaluable support on my path to completing this work. I would also like to thank my colleagues who have always encouraged me and been willing to lend an ear when I needed feedback.

Finally, I must express my very profound gratitude to my parents, Glen and Sharon, for their constant encouragement and support throughout my years of study, and to my wife, Jess, for her incredible patience with my many late nights working and endless discussions of my research.

This accomplishment would not have been possible without all of you. Thank you.

CONTENTS

FIGURES

TABLES

ABSTRACT

Powerful, handheld computing devices have proliferated among consumers in recent years. Combined with new cameras and sensors capable of detecting objects in three-dimensional space, new gesture-based paradigms of human computer interaction are becoming available. One possible application of these developments is an automated sign language recognition system. This thesis reviews the existing body of work regarding computer recognition of sign language gestures as well as the design of systems for speech recognition, a similar problem. Little work has been done to apply the well-known architectural patterns of speech recognition systems to the domain of sign language recognition. This work creates a functional prototype of such a system, applying three architectures seen in speech recognition systems, using a Hidden Markov classifier with 75-90% accuracy. A thorough search of the literature indicates that no cloud-based system has yet been created for sign language recognition and this is the first implementation of its kind. Accordingly, there have been no empirical performance analyses regarding a cloud-based Automatic Sign Language Recognition (ASLR) system, which this research provides. The performance impact of each architecture, as well as the data interchange format, is then measured based on response time, CPU, memory, and network usage across an increasing vocabulary of sign language gestures. The results discussed herein suggest that a partially-offloaded client-server architecture, where feature extraction occurs on the client device and classification occurs in the cloud, is the ideal selection for all but the smallest vocabularies. Additionally, the results indicate that

for the potentially large data sets transmitted for 3D gesture classification, a fast binary interchange protocol such as Protobuf has vastly superior performance to a text-based protocol such as JSON.

Chapter 1

INTRODUCTION

1.1 Background

Recent years have seen the widespread proliferation of powerful, affordable computing devices to an ever-growing set of users. Frequently, these devices are hand held and capable of fast calculation far beyond the capacity of desktop computers of just a few years ago. Along with the development of new sensing technologies such as capacitive multi-touch displays [Chang10] and cameras capable of sensing 3D depth, many new paradigms of physical interaction with our computers have arisen.

Sensors such as the Microsoft Kinect have greatly expanded the ability for software to recognize gestures in three-dimensional space. Specifically, the Kinect and similar devices can provide depth data in addition to RGB video, making them suitable candidates to employ with software seeking to recognize the gestures of American Sign Language. Additionally, the increasing inclusion of faster CPUs and powerful graphics processing units (GPUs) in mobile devices may allow for computer vision approaches to 3D gesture recognition on even these small, portable devices.

In addition to the developments in personal computing devices and sensing technology, recent years have seen an explosion in the availability of inexpensive, elastic compute

platforms in the commercial public cloud [Mirash10, Furht10]. This significantly lowers the barrier to entry for new applications, greatly simplifies the construction of highly distributed systems, and allows software running on mobile devices to easily off-load computationally intensive tasks to more powerful servers. This ability to distribute workloads between mobile devices and network servers allows countless mobile applications to very quickly perform tasks some modern desktop computers cannot complete on their own.

One widely known example of this strategy exists in the "virtual assistant" application that runs on modern smartphones such as Apple's Siri, Microsoft's Cortana, and Google Now. These systems leverage automatic speech recognition (ASR) to allow users to interact with the assistant using only speech. To accomplish this, the mobile device records and encodes audio spoken by its user but offloads much of the necessary processing to cloud services. The cloud services then interpret the audio input and return a transcribed result, usually based on large, robust models, providing a much more intelligent experience than the mobile devices could provide in isolation.

1.2 Problem Statement

Developing software applications to assist sign language communication with deaf people is an important area of work. Applications of this type could be used to aid deaf people so that they can interact with computer systems using sign language. Translation systems built on automated sign language recognition (ASLR) could assist

communication between deaf people and people who may not know sign language. Additionally, ASLR systems could be used to aid in the teaching of sign languages.

Some work has been done on ASLR, but it is not a fully mature knowledge area. Ong and Ranganath provided an excellent survey of the ASLR state of the art, including a review of major contributions as well as an analysis of neglected areas and suggestions for future work [Ong05]. Fortunately, ASLR has many similarities to automatic speech recognition (ASR), so it may serve as a useful reference point for designing ASLR systems as well as analyzing their performance. Herein, literature relevant to ASLR and ASR is reviewed, the gaps relevant to the proposed work are identified, and further research in the area is proposed.

Chapter 2

LITERATURE REVIEW

2.1 American Sign Language Recognition

American sign language, and sign languages generally, consist of manual and non-manual signing gestures. Manual signing consists of gestures isolated to the hands and arms. Non-manual signing encompasses broader movements of the head and torso as well as facial expressions. Manual signs can typically convey most of the lexical meaning in a sentence. Additional details are then provided by the non-manual aspects of the sign. These details may include intonation, verb tense, or intensity of action. While an ideal ASLR system would incorporate both manual and non-manual gestures, doing so is non-trivial. Especially in a mobile environment, accurate capture and analysis of non-manual signing may prove extremely difficult. Despite the constrained vocabulary afforded by this limitation, many useful applications may still be constructed using only manual signs.

The basic components of a manual sign consist of the shape, orientation, location, and movement of the hand, including both the palm and fingers. Perlmutter considers signs to be comprised of two segment types: position and movement [Permutter92]. Within these segments there are also secondary movements which can be seen as “internal movements” of the fingers relative to the hand. For accurate classification, an ASLR

system must be able to model both the static and moving components of signs. Additionally, when signs are composed to form sentences, there is movement between the individual signs that are not part of a sign and do not actually convey any meaning. This is similar to speech in the sense that the sound of a word may be affected by the words preceding and following it in a sentence. Finally, some signs may involve positions where some fingers obscure others, or if both hands are involved, one hand may partially obscure the other. More generally, signs are not flat but exist in three-dimensional space, so this must be accounted for when capturing data for an ASLR system.

2.2 Sign Gesture Capture and Classification

Computer vision and direct capture using gloves [Ong05] are the primary methods of capturing hand gesture data for ASLR. Generally, computer vision is a more desirable approach because it does not require specialized equipment and the user does not have to wear a special device to use the system. As such, a large body of research exists using computer-vision based approaches. With vision approaches, the two primary concerns related to ASLR are tracking of the hands and feature extraction. In regard to hand tracking, usually the full upper body of the signer needs to be in the camera's field of view. Using 2D video from a standard camera usually requires some restriction on the background and the clothing worn by the signer. Three-dimensional video using stereo cameras can overcome many of these limitations at the expense of greater computational cost. Critical features for detecting full signs include hand position (relative to the body),

shape, and orientation. Additionally, motion trajectories of the hands may be useful for some classifications. An in-depth discussion of tracking and feature extraction can be found in the survey by Ong and Ranganath [Ong05].

Upon feature extraction, signs are generally classified by either taking a sign as a whole and using a single classification step, or by breaking the sign into multiple components, classifying each component, and making a final classification based on the results of the components. The primary classification methods in the literature are neural networks and hidden Markov models (HMM) [Ong05]. Both methods have been shown to yield good results, but frequently each excels at different types of signs. Neural network-based approaches are frequently well-suited to classification of non-moving signs. HMMs, which are good at classifying time-series data, are best suited to dynamic, moving signs and series of signs formed into sentences.

2.3 Recent Work

While a consumer-ready mobile platform for sign language gesture recognition is not yet available, a large body of work exists on the problem of accurately interpreting sign language gestures in real time. Lichtenauer et al. demonstrated a system using stereo video cameras for gesture recognition in ASL by detecting skin as well as hand and head position in the video [Lichtenauer07]. Impressively, they were able to achieve 95% accuracy on 120 distinct signs performed by 70 people. The authors present a novel method for sign language gesture classification: acknowledging that many current

approaches use either Hidden Markov Models (HMM) or Dynamic Time Warping (DTW), they propose a new variation on the DTW approach. Using two cameras in order to collect 3D information, reference samples were taken for 120 different signs. For each of these a Bayesian binary classifier was then trained using samples taken from 70 persons, using DTW to normalize for different time-lengths and feature distribution within the gesture. Additionally, 50% Winsorization was performed on the training data to mitigate the impact of noise. Winsorization is a statistical technique, similar in effect to clipping in signal processing, that sets all outliers to a specified percentile of the data. In this case, all data below the 25th percentile was set to the 25th percentile and data above the 75th percentile set to the 75th percentile. These weak classifiers were then combined to form the main sign classifier, choosing the sign with the highest correctness. A 7-fold cross validation yielded a 95% true positive rate and 5% false positive rate. Additionally, the same data were tested using an HMM with 40 states using outcomes normalized by sequence length. This resulted in more than double the false negatives. Finally, the DTW Bayesian combined-classifier approach returned classification results within 50 ms, which is suitable for real-time applications.

Phadtare et al. presented a new method for feature extraction of hand and finger position data for static signs [Phadtare12]. To test their algorithm, they captured gestures using a Microsoft Kinect and processed the video with the open-source OpenNI library. Their work focuses on two sets of features: hand shape and palm orientation. The palm orientation is determined by finding the location of the wrist joint in the image as well as the contour created by the outer edge of the palm. The equation for the plane of the palm

is determined from these features. The core of this work lies in the hand shape detection algorithm. The authors proposed a three-dimensional extension of the Belongie shape context classification algorithm [Belongie02] which traditionally works in two dimensions. *N* points are sampled from the training shapes along the surface of the plane. The shape context is then constructed by computing the radial distance, radial angle, and altitude between each point and all the other sample points, and *K*-bin histograms of the distances are generated for each point. These histograms then constitute the trained model. Classification using this model involves a form of nearest-neighbors algorithm using the Chi-square distance metric. The authors tested their proposed algorithm using a set of 40 hand shapes. They report that the algorithm fails to differentiate shapes which only differ by slight variations in finger position. However, they report 20 shapes correctly classified and 10 shapes classified incorrectly but misclassified to a shape that was highly similar. The authors proposed increasing *N* and *K* to increase the accuracy of their approach. Finally, they noted that their algorithm is easily parallelizable and can take advantage of multi-core CPUs or even GPUs using technologies such as CUDA.

Kumarage et al. proposed new algorithms for computer-vision based sign language recognition to significantly decrease the compute intensity and allow for parallelism in the gesture processing [Kumarage11]. Their approach involved using combined learners to separately classify samples based on static features and movement features. The starting image, ending image, and movement sequence can be processed in parallel, increasing performance of the system. Additionally, by processing the starting image separately, the possible candidate classes were narrowed significantly, increasing the

performance of the final classification. For the still images, features were extracted to describe the shape and position of the hand and fingers. These features were then compared to training data stored in a database and possible candidate classes were assigned weights, although the authors are not clear about how these weights were generated. For motion classification, only the position of the hand was taken into account. Points were sampled along the hand's trajectory, and a least-squares approach is used to generate a best fit curve for the trajectory. The coefficients of the resulting polynomial were then compared to candidate classes in the database and weights were also generated. These weights were then combined to determine the final class for the gesture. While the specific classification algorithms used in this work do not seem novel, significant performance benefits from parallelizing the tasks were realized.

In addition to video feed, much work on automated sign language recognition uses features extracted from other 3D sensors. In work by Chuan et al., data are collected instead using a new 3D sensor called Leap Motion [Chuan14]. The Leap Motion controller includes an SDK with high level APIs. These APIs provide access to many features describing the position and movements of the hand and fingers. In this work, the authors demonstrate methods for deriving more meaningful features from the base feature set: average distance, which describes the average movement of all fingers between frames; average spread, which estimates the spread of the palm by averaging the distance between adjacent finger tips; and average tri-spread, which estimates the average triangular area between adjacent fingers. Additionally, several features are derived for the fingers which provide information about the finger relative to the palm instead of the

finger's absolute position within the frame. The data set studied in the work was comprised of the 24 static signs of the American Sign Language alphabet. The classifiers used were k-nearest neighbors (k-NN) and support vector machine (SVM). The highest performance reported was 84.5% with k=169 using the KNN classifier, with an average performance of 72.78% with k=7. The highest performance reported for the SVM classifier was 83.39% using the Gaussian RBF kernel, with an average performance of 79.93%.

Elakkiya et al. proposed a machine learning-based system for recognizing sign language gestures [Elakkiya14]. In their approach, the system does not use predetermined signs stored in a database for classification. Instead, it uses supervised learning to learn new words and phrases and gain feedback from the user on its interpretation of gestures. This approach is similar to many speech recognition systems and could lead to the development of a highly flexible and scalable system for sign language recognition.

Finally, the body of research on sign language gesture recognition is not confined to the English language or American Sign Language. Much work has also been done in this area in other languages, including but not limited to Indian, Korean, Arabic, Malay, Chinese, and Japanese [e.g., Min07, Swee07, Wang05, Lu97, Shanableh07, Baranwal14].

2.4 Considerations for Mobile Devices

While modern mobile devices have grown extremely powerful in recent years, user expectations have also grown. Many of the services users have come to expect require compute-intensive processes. These processes can present performance and energy consumption challenges on the mobile platform, so the most intensive aspects of these tasks are commonly offloaded to cloud services to increase performance and conserve energy. Automatic speech recognition (ASR) is one such service that is now frequently implemented using the cloud, including that seen in virtual assistants such as Apple's Siri and Microsoft's Cortana. Very early in the development of these types of systems, Rose et al. prototyped a system using server-based recognition to implement ASR on mobile devices [Rose01]. Their work highlighted several challenges inherent in these systems, including limited processing power on the device, energy consumption issues, and network bandwidth concerns for achieving real-time performance.

Automated speech recognition has much in common with automated sign language gesture recognition. In both types of systems, there is grammatical structure, and the data are generally treated as a time series, which lends to processing with HMMs. Especially for systems with very large vocabularies, classification using an HMM can be very computationally expensive. Due to this, Veitch et al. demonstrated the acceleration of ASR classification using general-purpose GPU computing [Veitch11]. They demonstrate that certain optimizations of the HMM algorithm, specifically the Gaussian calculation, can allow for massive single-instruction, multiple-data (SIMD) parallelism. In addition to

implementation on a GPU, the authors also implemented the algorithm using field-programmable gate arrays (FPGA) and compared the results. They also investigated the effect of parallelizing different aspects of the algorithm: one thread per model, one thread per mixture, or one thread per coefficient. The result showed that one thread per model resulted in the most efficient use of the GPU and the least memory transfer between GPU and CPU memory. With the optimized GPU implementation, the authors report the fastest performance to be 3.75 times faster than real time (10 ms), or 2.6ms per frame. They also report that this result is a 10-fold speed up from their optimized sequential CPU implementation. This significant speedup reduced classification latency to a level that is barely, if at all, perceptible by users. Keeping such low latency is crucial to providing near real-time interaction in such a system.

Chang et al. also prototyped a system for cloud-assisted ASR for mobile devices [Chang11]. Their approach attempted to accelerate performance and increase accuracy by incorporating a learning/training element for highly tailoring recognition to the user's voice. While many speech recognition systems are user-independent to make training unnecessary, this may sacrifice the accuracy of the system. In this work, the authors create a prototype that is initially user-independent but allows training over time to incorporate some user-dependent functionality. Additionally, their system anonymously uses this training data to improve the system for all users. To this end, the system must be "cloud-assisted" because cloud storage is abundant, relatively cheap, and allows sharing the model between all users. The cloud additionally provides the benefit of higher performance computing resources than those available on mobile devices. However, a

basic model was still stored on the mobile device to allow limited functionality even in a limited-network environment. This prototype system was able to greatly increase performance and accuracy relative to commodity ASR systems, leading to much greater user satisfaction and demonstrating the benefits of distributing workloads when cloud resources are available.

Nicholson and Noble conducted a more general investigation into network considerations on mobile devices [Nicholson08]. Acknowledging the resource constraints of mobile devices and the widely changing network environments of mobile users, they sought to develop a method of predicting future network connectivity. Using such a prediction, mobile applications could make more intelligent choices about network usage to optimize performance and energy usage. A system called BreadCrumbs was developed to track the performance of access points encountered at various locations visited by the user. At discrete timer intervals, the system scanned and tested all available access points within its immediate vicinity. As the user moved over time, changes in network performance were used to generate a second-order Markov model. This model then allowed the system to predict future network performance as the user changed location. Using only a short training period of a week, the system was able to accurately predict future bandwidth within 10 KB/s half the time and within 50 KB/s 80% of the time. Using these predictions, the authors were able to demonstrate improved performance and reduced battery consumption in several sample applications. This work concretely demonstrates the critical role network availability and usage may play in ASR applications.

Finally, Cuervo et al. proposed a system called MAUI, which enables offloading code execution from a mobile device to remote infrastructure [Cuervo10]. Included in this work, the authors demonstrate their system using a sample face-recognition application, which has similar compute demands to an ASLR system. Their system leverages the portability of the .NET Common Language Runtime (CLR), which uses an intermediate language and Just-In-Time (JIT) compilation to enable execution on different CPU architectures. The CLR also provides rich reflection capabilities. Using these capabilities, MAUI generates proxies for methods that allow remote execution, inspects application state at runtime, and transparently executes these methods remotely as needed. The system decides dynamically whether to use remote execution based on battery and network conditions towards a goal of minimizing energy consumption while simultaneously increasing application performance. In designing the system, the authors also studied the effects of network latency on energy consumption. They found that higher latency, such as that experienced on slower cellular connections, dramatically increased energy consumption, potentially negating the benefits of remote execution. Latency issues notwithstanding, the authors demonstrated very large gains in performance and reduction in energy usage when applying their system to the face-recognition application. These results show that offloading compute-intensive work may yield significant benefits in an ASLR application.

2.5 ASR Architecture

In 2008, Zaykovsky reviewed the current state of the art for mobile ASR systems and outlined three architectural possibilities for creating a mobile ASR system and details the primary concerns of each as well as the tradeoffs that exist between them [Zaykovsky08]. The primary tasks that must be accomplished by such a system are the capture of speech audio, feature extraction from the audio, and performing an ASR search to determine the most likely sequence of words that produced the audio input, usually using a Hidden Markov Model and Viterbi search.

The primary architectural approaches to this type of ASR system consist of Embedded Speech Recognition Systems, Network Speech Recognition Systems, and Distributed Speech Recognition Systems. The embedded approach consists of performing the entire set of ASR tasks on the mobile device using only the resources locally available. The Network Speech Recognition approach streams speech audio data to servers over the network, and these servers perform both feature extraction and the ASR search. Finally, the Distributed Speech Recognition approach combines the first two approaches, performing feature extraction on the mobile device, streaming the feature data to servers over the network, and performing the ASR search on these servers.

Embedded ASR systems, as shown in Figure 1, have the distinct advantage of not being dependent on a robust network connection. As discussed in [Zaykovsky08], the device acts in isolation and performs the entire ASR process independently. Implementation of

an embedded ASR system is becoming more attainable as mobile device hardware becomes more powerful. However, as users continue to expect systems to grow more sophisticated with the hardware, many of the constraints noted in early embedded ASR systems are still highly relevant. Primary among these constraints are limited storage, primary memory, execution speed, and battery capacity. Storage is quickly becoming a smaller concern, and algorithms can be optimized to somewhat overcome limitations in memory and processing power. Despite these possible optimizations, embedded ASR is best suited to the most powerful mobile devices and systems that make use of limited vocabularies.

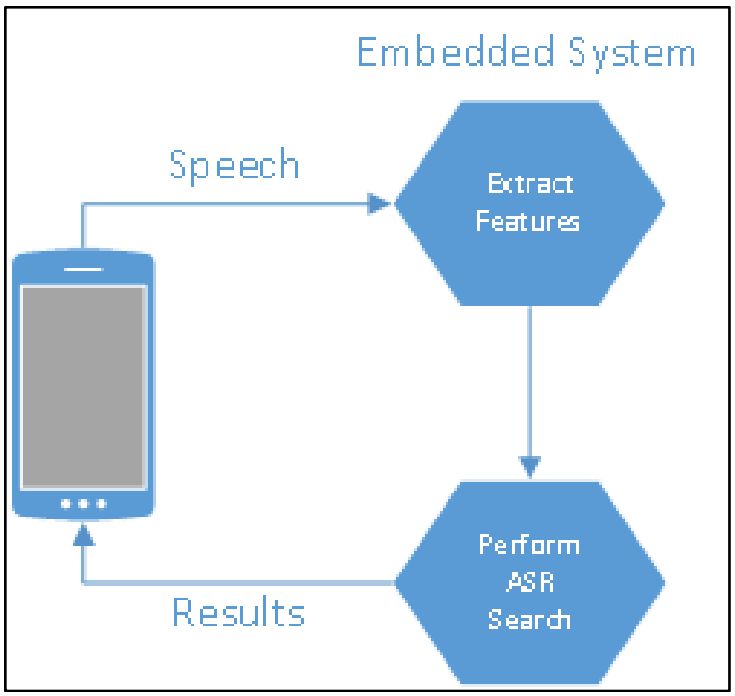

Figure 1: Embedded Speech Recognition Architecture

The second possible architecture, Network Speech Recognition, as shown in Figure 2, removes the constraints imposed by limited mobile device hardware resources and allows

applications to use state-of-the-art feature extraction and ASR search implementations executed on powerful servers. Raw or compressed audio data is streamed over the network to backend servers. On these servers, features are extracted, and ASR search is performed. Results of the ASR search are then streamed back to the mobile client over the network. More details can be found in [Zaykovsky08].

Another principal advantage of moving the processing away from the mobile device is that very large vocabularies may be used with the large storage available to the servers. Unfortunately, these systems also tether mobile devices to the network and ASR capability becomes limited if not impossible without a robust network connection since all captured audio data must be streamed over the network. Depending on the specific design of the system, this limitation can be somewhat overcome by certain audio encoding and/or compression algorithms to minimize network bandwidth requirements. Encoding and compression may inadvertently lose data important to the feature extraction process and result in less accurate ASR search results.

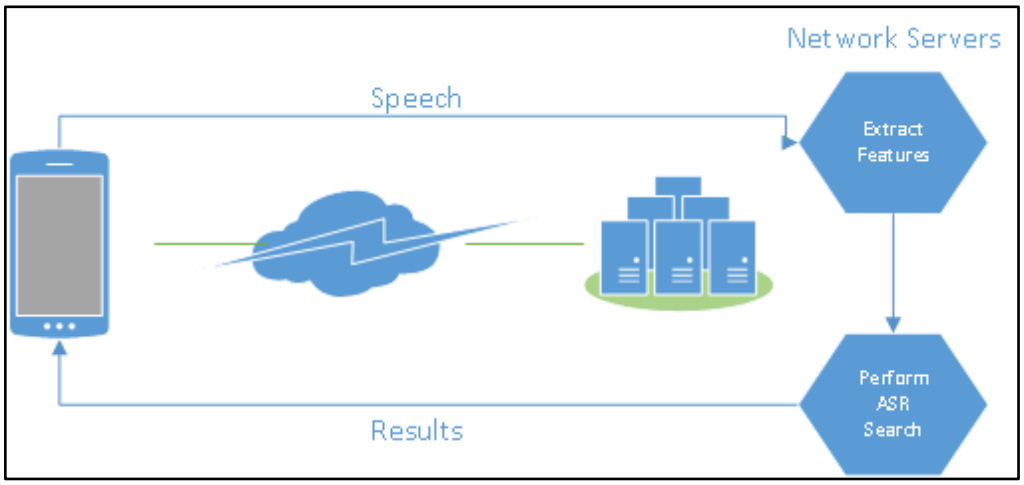

Figure 2: Network Speech Recognition Architecture

The final possible architecture, Distributed Speech Recognition outlined by Zaykovsky as shown in Figure 3, leverages the resources of both the mobile device and network servers to accomplish the ASR task. An overview of the distributed system design is provided in [Zaykovsky08]. In this type of system, feature extraction is performed on the mobile device, eliminating the high bandwidth requirement for streaming raw audio or the data loss incurred through compression. Extracted features are then streamed over the network to servers that perform ASR search.

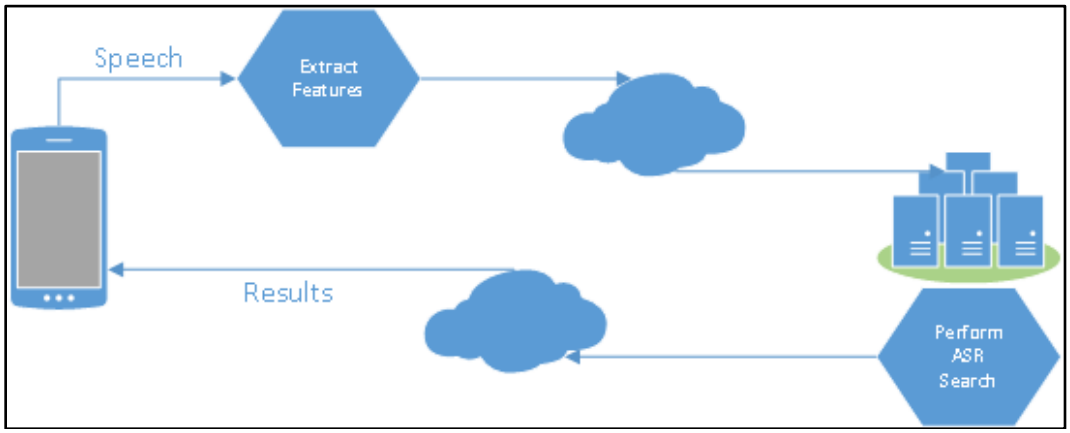

Figure 3: Distributed Speech Recognition Architecture

Rose and Arizmendi detail many common problems encountered with Distributed Speech Recognition systems and describe a client-server ASR framework to make optimal use of available resources and greatly boost performance, measured by both speed and accuracy, in a production ASR system [Rose06].

2.6 Conclusions

As previously outlined, there are three broad concepts involved in designing and constructing an automated sign language interpretation system for mobile devices: 3D sensing, sign language interpretation, and cloud-offloading of compute-intensive interpretation tasks. Each of these represents a grouping of more specific, related concepts. Within each group these concepts sometimes support each other and sometime are in conflict.

The first major concept is the issue of 3D sensing. Under this umbrella are several other concepts. The first of these is the type of sensing device used. Some researchers focus on sensors worn by the user or sensors in a device held by the user, while others detect "in-air" gestures using cameras and computer vision techniques. Another concept involved in 3D sensing is the interpretation of motion data and the recognition and categorization of gestures. Increasingly, this involves the use of neural networks but may also use a variety of other machine learning techniques. A final concept involved in 3D sensing is related to the user experience and determining gestures that are both natural and meaningful to the user as well as easily interpreted and identified by a computer system.

The second major concept identified is automated sign language interpretation. Within this larger concept there are also several related, more specific concepts. The first concept identified is related to the type of data captured for sign language interpretation and the ideal sensors for capturing such data. A second concept is that of interpretation and

identification of signs. This concept has two components. The first of these is the determination of the algorithm used, usually using a machine learning technique. The next of these is optimization of the chosen technique to increase performance and reduce compute requirements.

The final major concept is the need to use the cloud for the compute-intensive processing required by many of the machine learning techniques employed in gesture recognition and sign language interpretation. Within this area are also several related ideas. The first of these are the inherent limitations of mobile devices in terms of processing power and energy availability. Another concept is the effect of network bandwidth on performance as a distributed system is introduced. A final concept is the need to determine an optimal distribution of processing between the mobile device and cloud services.

Significant research has been conducted targeting each of the three major concepts identified. However, there is a dearth of work to integrate these concepts towards a mobile, real-time, automated sign language recognition system. The framework outlines within each of these concepts the key areas related to the development of such a system, as well as the inherent tensions between some of these concepts, such as the conflict between energy conservation and real-time performance.

Chapter 3

RESEARCH DESIGN

3.1 Questions and Objectives

The primary objective of this thesis is to create a proof-of-concept automated sign language gesture recognition system using cloud resources, demonstrating and evaluating various architectural strategies for implementing such a system using a mobile device and the cloud. To the best of our knowledge, no such system has yet been developed. Additionally, this research seeks to generate empirical data to support an analysis of the tradeoffs between the different architectural options and draw conclusions about the factors that might support choosing one option over another. The prototype system was created with three architectural variants, following the embedded, distributed, and network architectures outlined by Zaykovsky, and key system metrics were observed in each setting. Zaykovsky's language describing the architectural variants is somewhat dated in the context of cloud computing. In light of this, updated terms are used henceforth: "client-only", "partially-offloaded client-server", and "fully offloaded client-server", reflecting the amount, if any, that compute-intensive tasks are offloaded to cloud resources. Additionally, two serialization/interchange protocols were used for transmitting data over the network for the partially-offloaded and fully-offloaded client-server architectures to observe the effects of data interchange format for large payloads.

3.2 Methodology

For the purpose of this research, the term "application" describes all the components required to implement the features of a basic ASLR system. Depending on the architecture being examined, this may include only software running on a client device, or it may also include software running on a network server (possibly in the cloud). A prototype is created for each of the three architectures being studied. Since this study examines the effect of distributing ASLR processing across multiple systems, algorithms are implemented as similarly as possible, whether they are running on a mobile device or on a network server. ASLR recognition tasks are conducted using the three prototypes, and key metrics are recorded during the use of each prototype. The metrics being examined include application response time, mobile device CPU usage, mobile device memory usage, and network bandwidth usage.

Experiments are performed to compare the performance of client-only, partially-offloaded client-server, and fully-offloaded client-server ASLR architectures. The Kinect V2 for Windows sensor is used to capture gesture data. Due to limitations of the Kinect SDK, data from the sensor are recorded and replayed on a resource-limited device such as an Intel Compute Stick. The Kinect for Windows SDK is used to provide useful features from the raw sensor data feed. Development and testing of the prototype system is done on a 2.8 GHz Intel Core i7 MacBook Pro.

The implemented ASLR algorithm is Hidden Markov Model, provided by the well-known library Accord.NET for the .NET Framework. The model is trained using data from the signs listed below with at least 6 instances of each sign. The signs used comprise 11 categories and are listed in Appendix A.

In each experiment requiring the use of a network server, the server application is constructed using Microsoft's open-source ASP.NET web technologies. The application runs in the Microsoft Azure public cloud to simulate the environment of a real-world application. The server application code is instrumented using the built-in instrumentation tools available with the .NET framework, as well as some custom wrappers around lower-level Windows APIs. These libraries provide low-overhead, high-resolution stopwatches and performance counters for measurement of memory usage.

The next chapter discusses in depth the extraction of useful features from data provided by the Kinect API and the training of Hidden Markov models for ASL gesture recognition. Following this is a detailed discussion on the implementation of the three architectures examined by this study.

Chapter 4

HIDDEN MARKOV MODEL TRAINING

In this chapter, the training of a Hidden Markov Model to classify American Sign Language gestures is discussed. Recordings were made using the Kinect Sensor and Kinect Studio software tool. After collecting these recordings, routines were created to extract derived features from the raw body-tracking data. These derived features were then used to train a Hidden Markov Model implemented by the open-source Accord.NET framework. The trained model achieved approximately 80% accuracy.

4.1 Raw Sign Data Collection

Recordings containing all available channels from the Kinect Sensor (including 1080p color video) requires and excess of storage space. To reduce the amount of storage space required, recordings excluded the high-definition color and audio channels, since these are not necessary for obtaining body-tracking information. Instead, only the IR/depth and body-tracking channels were retained. This reduced the file size per raw recording to approximately 150 megabytes.

4.2 Feature Extraction

The raw body-tracking data available via the Kinect SDK includes a wealth of skeletal data points. Included in the available set are positions and angles for major joints: shoulder, elbow, and wrist; upper, middle, and lower spine; head and neck; hips, knee, ankle, and foot; and hand tip and thumb positions. The Kinect SDK provides several additional features such as body lean and hand tracking confidence. These features, in their raw form, are not useful for training a Hidden Markov Model that can detect American Sign Language gestures.

In order to train a useful model, several transformations were required to arrive at suitable derived features. The first transformation involved normalization of the raw features for consistency between the recordings from people of different body sizes and to account for slight variations in distance from the sensor during recording. The normalization consisted of two phases. First, the joint positions were repositioned such that the mid-spine position was a reference point at the zero-coordinate in all three dimensions. After repositioning, the joint positions were scaled. A reference scale was created by computing the Euclidean distance between the top and base of the spine. Each point was then divided by the reference distance.

After normalization, fourteen features were derived from the normalized joint positions. These consisted of three inter-joint distances, two joint angles, and the hand area on each side of the body. The features were as follows: 1) the Euclidean distance between the

wrist joint and the mid-spine, 2) the Euclidean distance between the elbow joint and the mid-spine, 3) the angle of the elbow joint, 4) the angle of the shoulder joint, 5) the wrist-to-wrist distance, 6) the hand area, and 7) the angle between the thumb and hand tip. These features are shown in Figure 4 below.

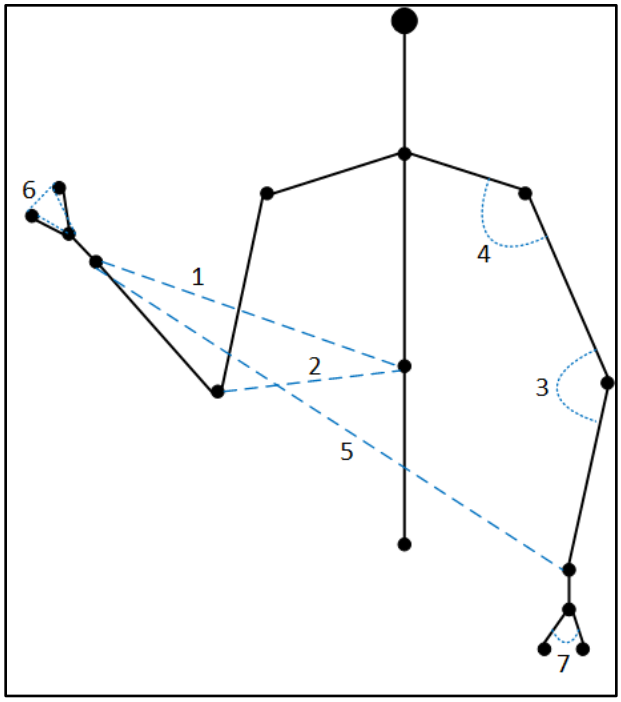

Figure 4: Diagram of features derived from Kinect skeletal tracking data.

Finally, after the derived features were computed, a final smoothing step was undertaken to reduce noise in the feature set. The smoothing was accomplished by averaging the features from each frame with the corresponding features in the prior and upcoming frames.

4.3 Model Training and Validation

For this experiment, a Hidden Markov classifier was trained by creating a Hidden Markov Model for each of the ten recording signs, using the derived features described above. An off-the-shelf implementation was used from the open-source Accord.NET library. This library provides several classes to facilitate training of a Hidden Markov classifier.

First, an instance of the `HiddenMarkovClassifier<TDistribution, TObservation>` class was initialized. The TDistribution generic parameter used in this case was `MultivariateNormalDistribution`, and the `TObservation` parameter was `double[]`. An instance of the `HiddenMarkovClassifierLearning` class was then used to train the classifier using the Baum-Welch learning algorithm. The learner was configured to parallelize using all the CPU cores available on the training machine (4 physical, 8 virtual).

For each sign, six recordings were collected. Five of each were used to train the model with one set aside for model validation. After training, a classifier was created that could differentiate the signs with approximately 80% accuracy. While a real-world system would ideally have significantly higher accuracy, for the purposes of this experiment – measuring architectural effects on performance – this level of accuracy was sufficient. Persisted to disk, this model required 140 kilobytes of storage.

Chapter 5

IMPLEMENTATION OF ASLR ARCHITECTURES

In this chapter, the details of each architectural implementation (client-only, partially-offloaded client-server, fully-offloaded client-server) are described. While each implementation is unique, code was shared between each architecture when possible in the interest of implementation efficiency as well as a reduction in the number of variables between each that could affect performance. All implementations were written in the C# language using the .NET Framework. The Visual Studio IDE was used throughout the implementation process for code editing, compilation, and debugging. After detailing the implementation of each of the three architectures, a discussion of performance instrumentation follows.

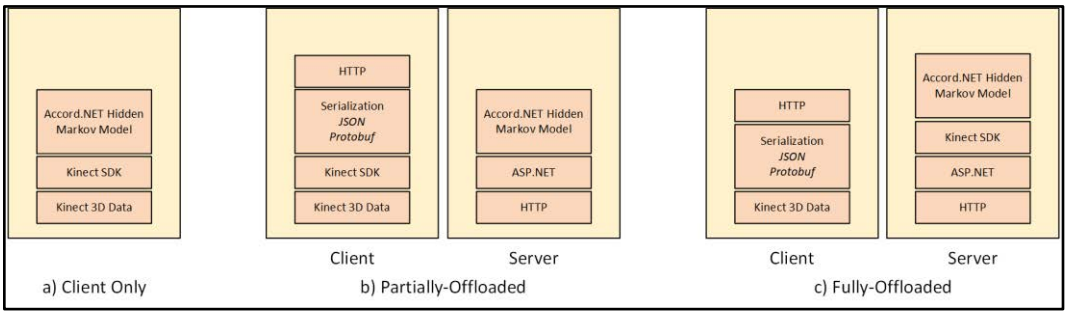

Figure 5: Architecture diagram of ASLR system.

A diagram of the overall system architecture is shown in Figure 5 above. The application first captures or replays sensor tracking data. Subsequently, useful features are extracted

from this raw data. In the client-only and partially-offloaded client-server architectures, this is done on the same client device where raw data capture/replay occurs. In the fully-offloaded architecture, the raw data are sent over the network and feature extraction is performed on a cloud server. These features are then used as parameters to the Hidden Markov Model (HMM) classifier, which provides a classification of which gesture was performed. In the client-only architecture, this step is also performed on the same client device. In the client-server architectures it is performed in the cloud service; in the partially-offloaded case, the extracted features are sent over the network to the cloud service. With this high-level overview of the system, the rest of this chapter will describe the details of each architecture.

5.1 Common Implementation Details

While incredibly useful, the majority of data types provided by the Kinect SDK are opaque. In order to control serialization, as well as the ability to stream Kinect tracking data to a cloud service where the SDK was not installed, proxy classes were created for all relevant Kinect SDK data types. This includes core geometric types like `PointF`, `Vector3`, and `Vector4`; the body-tracking types `BodyFrame`, `Body`, `BodyJoint`, and `JointOrientation`; and various enumerated types for body/joint tracking details: `TrackingState`, `TrackingConfidence`, `JointType`, and `HandState`. Each proxy type perfectly mimicked the corresponding Kinect type, but without any dependency on the Kinect SDK. Additionally, these proxy types were configured using C# attributes for JSON and Protobuf serialization.

A C# class was created to model the experiment for each of the three architectures being tested: `ClientOnlyExperiment`, `FullOffloadExperiment`, and `PartialOffloadExperiment`. Because of commonalities between each architecture, as well as "boilerplate" code required for each experiment, the classes modelling each experiment were derived from one or more abstract base classes containing the common behavior. The common base class, `AbstractExperiment`, is responsible for loading the Hidden Markov Model from the filesystem, initializing stopwatches and performance counters, and measuring the performance information of the algorithms implemented by its concrete subclasses.

5.2 Common Network Details

Inheriting directly from `AbstractExperiment` is the `NetworkedExperiment` abstract class. This class is responsible for the serialization of raw Kinect body-tracking data or derived features to JavaScript Object Notation (JSON) or Protobuf format for the experiments that rely on a network-connected cloud service. This class makes use of C# generics to implement a common serialization routine regardless of whether the concrete subclass is working with raw body-tracking data or derived features. In addition to reducing the amount of required code, this reuse also ensures no unintended performance differences due to serialization between the fully-offloaded and partially-offloaded client-server architectures.

5.2.1 Serialization Protocols

For the experiments that required data to be sent over the network, two different serialization methods were used so that the effect of serialization on performance for network-based architectures could be measured. The first serialization protocol used was JSON, a portable, text-based data interchange format based on the conventions of the JavaScript (ECMAScript) programming language. The JSON protocol [Json17] supports several primitive data types, as well as the structured data types *object* and *array*. It is a widely used data-interchange protocol, with client libraries for a multitude of programming languages and environments. The strength of the JSON protocol is its flexibility and, therefore, portability. For the experiments described here, the popular Newtonsoft Json.NET library was used to provide JSON serialization [JamesNK18].

The second serialization protocol used was Protobuf, a binary data-interchange format created by Google [Protobuf18]. While Protobuf aims for high portability, its primary aim is performance. The Protobuf protocol imposes more constraints than highly-flexible protocols like JSON and XML in the name of faster serialization/deserialization and smaller binary message footprint. A domain-specific language (DSL) is used to define Protobuf message specifications, from which serialization and deserialization code is generated. This is traditionally accomplished using *proto* files, containing the DSL, which are compiled with the *protogen* compiler utility. Various other utilities can then be used to generate message proxy code in common programming languages such as C++, Java, or C#.

An alternative method, used in this case, is to have the *proto* specifications and proxy code dynamically generated at runtime by a library. In .NET applications, this is accomplished using the Protobuf.NET library [mgravel18]. This library employs the reflection and runtime code-generation facilities of C# and the .NET Runtime to create *proto* specifications by inspecting C# attributes decorating members of "plain old CLR objects" (POCOs). While runtime reflection and code generation can be computationally expensive, the output is cached by the library, so the computational cost must only be paid the first time the code is executed within the process. This approach greatly eases the use of the Protobuf interchange format and makes for easier-to-read, more maintainable code.

5.3 Kinect Recording Playback

Finally, a core common feature shared among each experiment implementation is the replay of the Kinect SDK recordings. These recordings are stored on the local filesystem. In each case, the recordings are replayed using the Kinect Studio APIs, which provide the data for consuming applications via the Kinect service. Replaying the recordings using this method mimics having a real Kinect sensor connected during the experiment. The experiment code uses the exact same APIs as connecting to a physical sensor, and it receives data as if it were being collected in real-time from a sensor. In addition to this, by replaying the data via the Kinect service, real-world constraints are present, such as dropped frames due to poorly-performing application code. The file replay functionality

was abstracted into its own class, the `KinectFilePlayer`, which hides the low-level details and provides a clean API surface for the consuming code.

5.4 Client-Only Architecture

In the client-only architecture, all processing is done on a single device, with none of the workload distributed to the cloud. An Intel Compute Stick (Model STK2M364CC) was used as a stand-in for a mobile device, such as a smartphone. This device's CPU was an Intel Core m3-6Y30 processor, with 4 MB cache and a clock speed of up to 2.2GHz. The device had 4 GB of LPDDR3-1866 main memory, and 64 GB of embedded storage. The fairly limited hardware resources available on this model of Compute Stick make it comparable to many premium smartphones currently on the market, and the CPU in the compute stick performs similarly to the processor in high-end smartphones such as the iPhone 6s or iPhone 7.

The implementation of the `ClientOnlyExperiment` class was fairly straightforward. Inheriting from `AbstractExperiment`, the abstract `RunCoreAsync` method was overridden to provide the core functionality which was measured. In this method, features were first extracted from the body-tracking data provided by the replayed Kinect recording. After feature extraction, the Hidden Markov classifier was loaded from disk. Then, the extracted features were passed to the classifier to obtain a result.

5.5 Fully-Offloaded Architecture

In the fully-offloaded architecture, after reading the body-tracking from the Kinect recording, all processing was offloaded to a cloud service. The client device acted as a "thin client", doing no heavy processing of its own. As in the client-only architecture, the client device was an Intel Compute stick. The cloud service was an ASP.NET Core Web API Application, hosted in the Microsoft Azure public cloud. Specifically, the application was hosted in an Azure App Service plan at the B1 pricing tier. This tier provides one virtual CPU core, 1.75 GB RAM, and 10 GB storage.

5.5.1 Thin Client Implementation

The thin client of the fully-offloaded architecture was implemented by the `FullOffloadExperiment` class. Like the client-only implementation, the core logic was implemented by overriding the abstract `RunCoreAsync` method. In this case, the method was responsible for reading the raw body-tracking data from the Kinect recording, configuring an HTTP client to communicate with the cloud service, serializing the body-tracking data, and sending an HTTP request.

The HTTP client used was the `System.Net.Http.HttpClient` class, provided by the .NET framework. The body-tracking data were transmitted in the body of a POST request and were serialized as either JSON or Protobuf. To avoid doubled memory usage due to serialization, the output of the serializer was written directly to the request stream,

instead of using an in-memory buffer. This was accomplished using built-in framework classes: the high-level `ObjectContent<T>` class for JSON and the `PushStreamContent` class for Protobuf because there is no high-level built-in class suitable for the Protobuf implementation.

5.5.2 Cloud Service Implementation

The cloud service in the fully-offloaded architecture was responsible for deserialization of the body-tracking data from the HTTP request, feature extraction from the raw deserialized data, and classification using the Hidden Markov classifier. The service was implemented using the ASP.NET Core 2.0 Framework, exposing a single endpoint via an action method on a `Controller` subclass. The hosting web application was configured globally to handle deserializing HTTP request bodies as either JSON or Protobuf, depending on the Content-Type header specified in the request. This was accomplished using the Newtonsoft Json.NET for the JSON protocol and the Protobuf.Net and WebApiContrib.Core.Formatter.Protobuf libraries for the Protobuf protocol [WebApi18].

After deserializing the request body and validating the request, processing was delegated to the `GestureClassificationService` class. For the fully-offloaded experiment, this class was responsible for extraction of features from the raw body-tracking data using the `DerivedGestureFeatureExtractor` class. Upon extraction of features, the class then loaded the Hidden Markov classifier from disk, and

classified the gesture based on the derived features. After returning this classification to the controller, the controller wrote the classification to the HTTP response as JSON.

5.6 Partially-Offloaded Client-Server Architecture

In the partially-offloaded client-server architecture, after reading the body-tracking from the Kinect recording, the processing was shared between the client application and a cloud service. The client device extracted features from the raw data and sent only these derived data over the network. As in the client-only architecture, the client device was an Intel Compute stick, and the cloud service was an ASP.NET Core Web API Application, hosted in the Microsoft Azure public cloud using the same B1 pricing tier.

5.6.1 Client Implementation

The client of the partially-offloaded client-server architecture was implemented by the `PartialOffloadExperiment` class. As in the other implementations, the core logic was implemented by overriding the abstract `RunCoreAsync` method. In this case, the method was responsible for reading the raw body-tracking data from the Kinect recording and extracting features using the `DerivedGestureFeatureExtractor` class. Upon extracting features, the method was responsible for configuring an HTTP client to communicate with the cloud service, serializing the extracted feature data, and sending an HTTP request.

As in the fully-offloaded implementation, the HTTP client used was the framework-provided `System.Net.Http.HttpClient` class, and the feature data were transmitted in the body of a POST request, serialized as either JSON or Protobuf. The data were, as before, written directly into the request stream from the serializer output, to avoid an unnecessary increase in memory footprint.

5.6.2 Cloud Service Implementation

The cloud service in the partially-offloaded client-server architecture was responsible for deserialization of the derived feature data from the HTTP request and classification using the Hidden Markov classifier. The service was implemented using the ASP.NET Core 2.0 Framework, exposing a single endpoint via an action method on a `Controller` subclass. As in the fully-offloaded case, the hosting web application was configured globally to handle deserializing HTTP request bodies as either JSON or Protobuf, depending on the Content-Type header specified in the request. This was, as before, accomplished using the Newtonsoft Json.NET library for the JSON protocol and the Protobuf.Net and WebApiContrib.Core.Formatter.Protobuf libraries for the Protobuf protocol.

Upon deserializing the request body and validating the request, processing was again delegated to the `GestureClassificationService` class. For the partially-offloaded client-server architecture implementation, this class was responsible for loading the Hidden Markov classifier from disk and classifying the gesture based on the derived

feature data provided. As before, upon returning this classification to the controller, the controller wrote the classification to the HTTP response as JSON.

5.7 Performance Instrumentation

In order to assess system performance across the various architectures, several methods of measurement were used. To measure the wall time required to complete classification for a single gesture, the `System.Diagnostics.Stopwatch` class, provided by the .NET Framework, was used. This class provides a basic high-resolution stopwatch with a simple API to start, stop, and measure elapsed time. To measure CPU time, a custom class was implemented (described in section 5.7.1), since the .NET Framework does not have a built-in facility for measuring CPU time. To measure network and memory usage, Windows performance counters were used.

5.7.1 High-Resolution Stopwatch

The .NET Framework contains a built-in high-resolution Stopwatch class. However, this class is only useful for measuring "wall time" (i.e., real-world clock time) elapsed between start and stop of the stopwatch. While wall time measurement was useful for the purposes of this research, a high-resolution measurement of CPU time was also needed. The .NET Framework does not have built-in support for this measurement, but the Win32 API on Windows does provide this facility. To access this measurement from the .NET Runtime environment, an `ExecutionStopwatch` wrapper class was created using the

P/Invoke feature to call into the Windows kernel API. The class calls into Kernel32.dll, the Windows system library responsible for exposing the core Windows API, including functions for process and thread management. This class made use of two Windows API functions: one to obtain the native Windows process handle for the currently executing process, and a second to obtain the CPU time used by the process in both kernel and user space. For the purposes of this experiment, the kernel and user space times were added to obtain the total amount of CPU time used by the process.

5.7.2 .NET Performance Counters

The .NET Runtime and Windows provide the Performance Counter APIs [Performance18] for measuring various information regarding usage of compute resources by a specified process. Included in the available APIs are counters for measuring network and memory performance.

Because the .NET Runtime provides a managed memory environment with a garbage collector, it is difficult to measure memory performance in terms of absolute memory allocations. However, a useful metric in such an environment is a measurement of the number of garbage collections. The .NET runtime's garbage collector is generational, where heap objects are assigned to one of three "generations" based on the amount of memory used by the object as well as its lifetime. Older generations are reserved for large and/or long-lived objects, while younger generations are for small and/or quickly-collected objects. By measuring the number of collections in each generation, insight is

provided into both the amount of memory required by the process as well as how much memory churn occurs.

In order to measure the garbage collections for each generation, three different performance counters were used, each within the ".NET CLR Memory" category: "# Gen 0 Collections", "# Gen 1 Collections", and "# Gen 2 Collections". Specifically, the "Gen 0" counter measures the number of times the garbage collector has executed a collection of generation-zero objects. These collections occur when the amount of memory available in the generation-zero heap is insufficient to satisfy a new allocation. The objects remaining in the generation after collection are promoted to generation one. The "Gen 1" and "Gen 2" counters measure the number of times the garbage collector has executed a collection of objects in the generation-one and generation-two heaps, respectively. In depth information on the .NET Runtime garbage collector is available in the Microsoft documentation [GarbageCollection18].

To measure network usage, two counters were used in the ".NET CLR Networking" category: "Bytes Sent", and "Bytes Received". These counters measure the total number of bytes sent and received by all sockets within the AppDomain. Importantly, the AppDomain represents a lightweight, managed "process" within the .NET Runtime. As multiple AppDomains can exist simultaneously within the same Win32 process, these performance counters only measure network usage by the current AppDomain, not the entire Win32 process.

Chapter 6

PERFORMANCE ANALYSIS

The implementation of a prototype ASLR system was completed using client-only, partially-offloaded client-server, and fully-offloaded client-server architectures, using various data serialization protocols for each of the networked architectures. Each architecture/serialization combination was tested using HMM classifiers of increasing vocabulary sizes: 10, 17, and 23 sign language gestures. This system, as implemented, shows the potential performance bottlenecks of each architecture, which may provide useful guidance to the implementer of a production-grade ASLR system. The results show that each of the architectures may be suitable for certain use cases or scenarios, and a production-grade system may benefit from incorporating elements of each. Additionally, there was a notable performance difference between the JSON and Protobuf serialization protocols, highlighting the importance of the data interchange protocol when transferring the type of data required in an ASLR system.

For each architecture, and for each serialization protocol for networked architectures, the CPU, network, and memory performance of the system was measured using models of increasing vocabulary size. A single experiment run consisted of classifying a sample of each of the signs known to the model, using each of the architecture/serialization combinations. The experiment was run for 100 iterations.

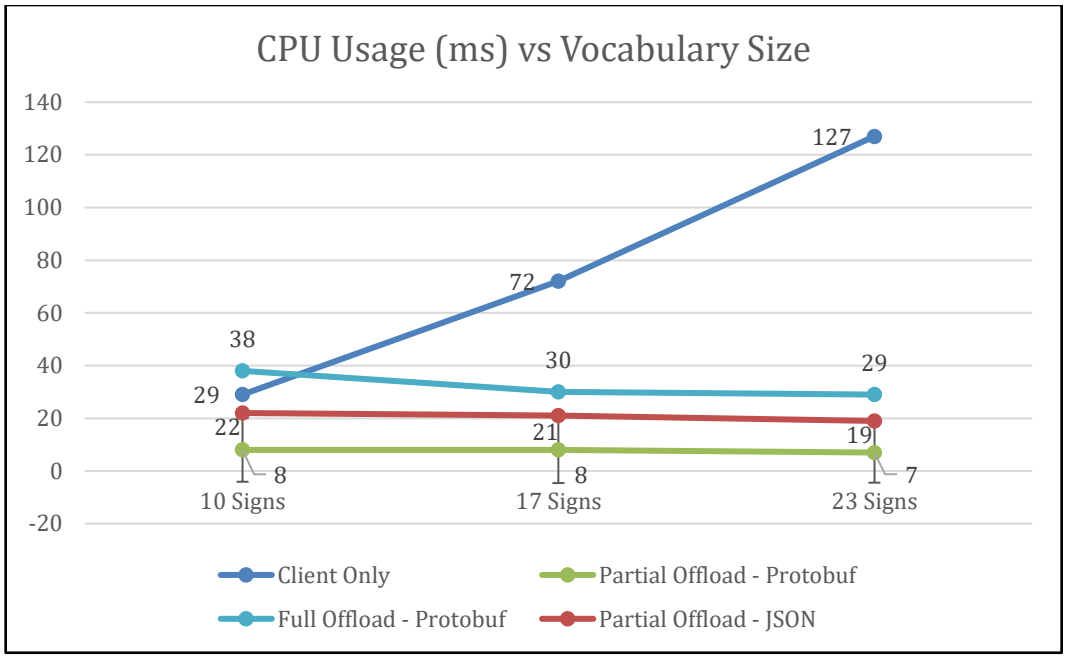

Figure 6: Average CPU clock measurements.

6.1 CPU Usage and Response Time

Figure 6 above shows the average CPU clock measurements for each run. Somewhat surprisingly, the client-only architecture performed very well with a small 10-sign vocabulary, completing feature extraction and classification in an average of 29 milliseconds. However, as the number of signs in the vocabulary increases, the classification time steadily increases. The 17-sign model completed classification in 72 ms CPU time, and the 23-sign model completed in 127 ms CPU time. This trend suggests that significantly larger classifier vocabularies could require considerably more CPU time, resulting in decreased perceived performance by the user, as well as greater energy usage. Finally, the fully-offloaded architecture using JSON serialization showed an enormous increase in CPU time, from 455 to 606 ms used.

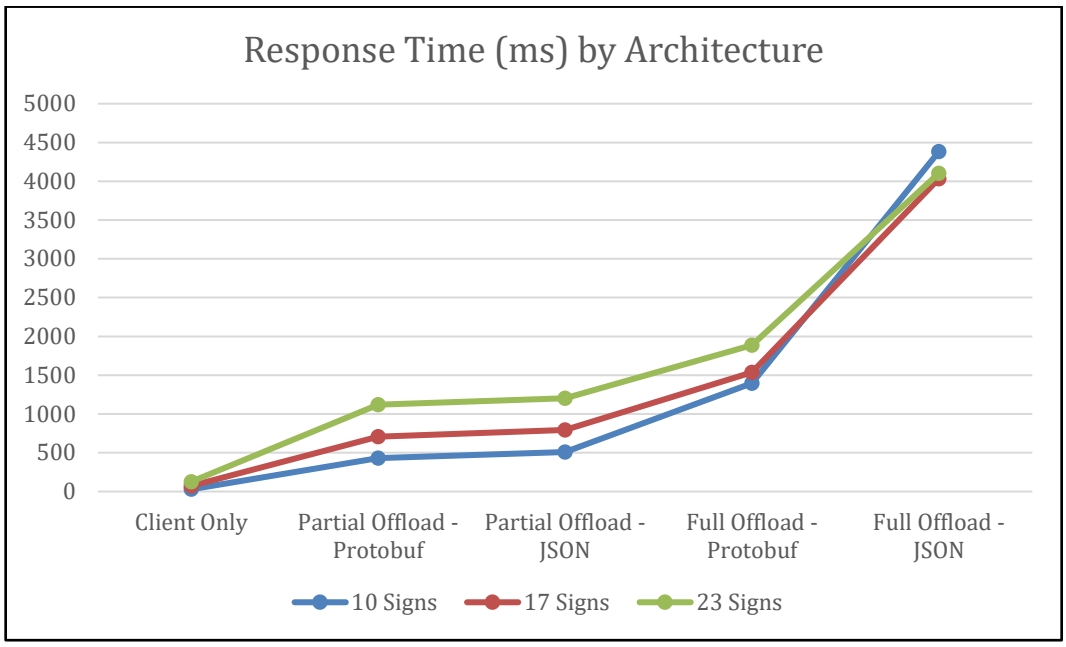

Figure 7: Average response time (wall clock) measurements.

Figure 7 above shows the average response time of the system for each architecture/serialization combination. The client-only case roughly matches the measured CPU time for the same case. The slightly increased wall time relative to CPU time is expected due to the regular context switching that occurs in a multitasking operating system. Distributing the classification to the cloud service showed a small increase in response time, with a maximum of about 1.2 seconds. Some increase over the client-only case was expected, due to network latency and data transmission time. The relatively small difference in the Protobuf and JSON cases is likely attributable to the small increase in CPU time in the partially-offloaded/JSON case.

The increase in response time as the number of signs in the vocabulary is likely due to an increase in CPU time for classification in the cloud service. While this trend would eventually present a problem with larger sign vocabularies, the cloud architecture allows for far more optimization than is possible in the client-only case. Strategies similar to the previously reviewed work by Veitch [Veitch11] – optimizations for massive parallelism of HMM classifiers – could be employed using pools of cloud resources to significantly increase the possible vocabulary size with minimal performance degradation.

Finally, the fully-offloaded scenarios had a significantly increased response time, with the most notable increase in the JSON serialization case. The fully-offloaded/Protobuf combination yielded classifications in 1.4-1.9 seconds, while the fully-offloaded/JSON case required 4.1-4.4 seconds to return a classification. Both of these results represent a significant performance degradation in terms of user-perceived system performance. This sizeable increase in response time is most likely due to the large data payloads required when sending raw body-tracking data across the network. Because of the large amount of data being transmitted, JSON serialization performs especially poorly, due to the redundant nature of the protocol. While compression could be applied to mitigate this effect, an increase in CPU usage would likely occur, somewhat negating the benefit of the compression.

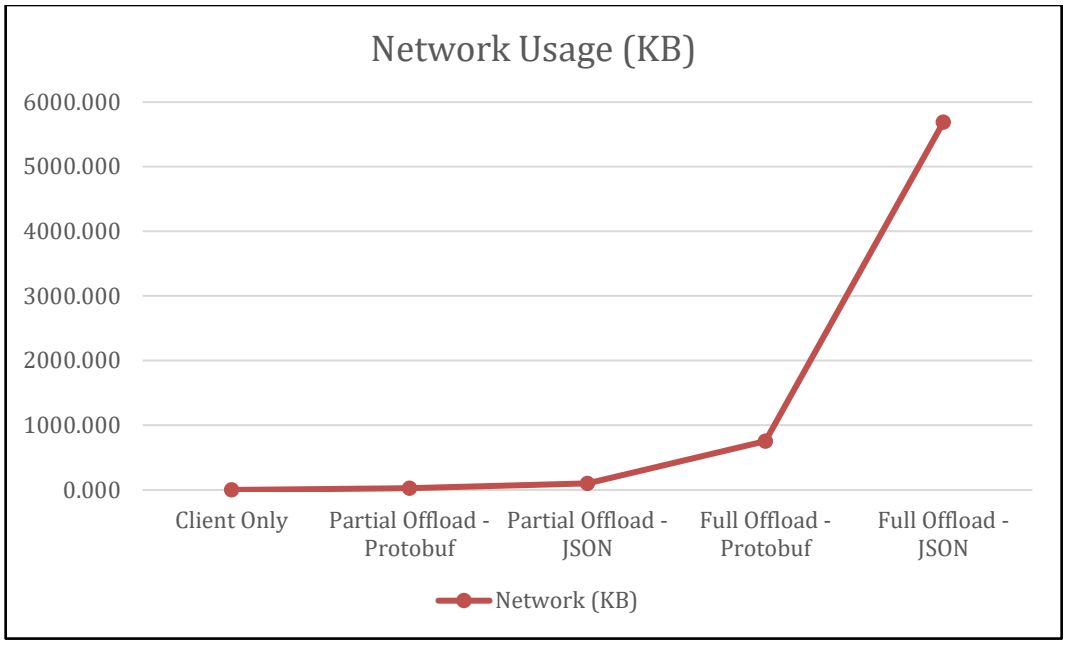

Figure 8: Average network usage measurements (total kilobytes).

6.2 Network and Memory Usage

Network performance was also measured for each architecture and serialization protocol. Figure 8 above shows the average network usage for each architecture/serialization combination. As the network payload is not affected by the number of signs in the model, the values shown are the average across each model size. The client-only case, by definition, had no network usage. The partially-offloaded client-server architecture had relatively low usage, 25.7 KB with Protobuf and 99.5 KB with JSON. The fully-offloaded client-server architecture required a significant increase in network usage. Using Protobuf, the fully-offloaded case used 753.5 KB of network bandwidth, while the JSON case used an enormous 5.68 MB. As discussed above in the context of response time, compression of the JSON request body could significantly reduce the required

network bandwidth. However, the extra CPU time required to compress the request may negate the benefit of reduced network bandwidth.

Finally, memory performance was measured for each architecture and serialization protocol. Table 1 below shows the aggregate memory performance measurements. Memory allocations, measured by the number of garbage collections (GCs) performed by the runtime, are shown here as the minimum and maximum number of collections observed, since varying system memory conditions affect the runtime's decision to perform a collection.

	Serialization	Minimum GCs	Maximum GCs
Client Only	--	0	6
Partial Offload	JSON	0	0
Partial Offload	Protobuf	0	2
Full Offload	JSON	136	383
Full Offload	Protobuf	0	3

Table 1: Aggregate memory usage by number of garbage collections (GCs).

The number of garbage collections performed by the runtime during the client-only experiment varied from none to six, which is a moderate number of collections. Somewhat surprisingly, both partially-offloaded cases as well as the fully-offloaded/Protobuf case had an even lighter memory footprint than the client-only case. This may indicate that a large part of the memory usage occurs in the HMM classification step rather than during feature extraction. By moving the HMM step to the server in both cases, the memory usage on the client device is reduced. In contrast, the fully-

offloaded/JSON case had an enormous memory footprint. Given the large data set transmitted in the fully-offloaded case and the repetitive nature of arrays containing JSON objects, it's likely many small, short-lived objects were allocated during the serialization process, contributing to the large number of collections.

Chapter 7

CONCLUSIONS

This research has examined the performance impacts of various architectural strategies for the implementation of a real-time automated sign language recognition system in a resource-constrained environment such as a mobile device. The results, sometimes surprising, have provided useful insight into which factors and design choices most impact the performance of such a system. Specifically, the results provide two noteworthy findings. First, the reasonably good CPU performance of the client-only architecture demonstrates that systems requiring a small vocabulary, such as command and control applications, could feasibly be implemented entirely on a mobile device or similar system, with no need for cloud resources. This is an encouraging result for systems that may need to function when there is no readily-available Internet access. However, in a system requiring a large vocabulary, performance in the client-only scenario may degrade as the vocabulary size increases.

Second, the effect of data payload size sent over the network had a large impact on system performance. The significant increase in resource usage across the board for the fully-offloaded client-server architecture compared to the partially-offloaded client-server architecture shows this clearly. This was further highlighted by the effect of the data interchange protocol in the networked architectures. Protobuf consistently required less

CPU time, network bandwidth, and memory allocation than JSON, in some cases outperforming JSON by an order of magnitude.

Overall, the overall best performance was shown by the partially-offloaded architecture using the Protobuf data interchange format. These results, together with a general knowledge of Hidden Markov classifiers, suggest that this would be the best choice in a real-world scenario with a model trained on a large number of signs. The performance impact of feature extraction and Protobuf serialization on the resource-limited device was small, and the cloud is ideally suited to algorithms like Hidden Markov classifiers, which perform well when highly parallelized.

Crucially, the prototype created by this work is, to the best of our knowledge, the first automated sign language recognition system to distribute compute-intensive tasks to cloud resources. This is an invaluable contribution – overcoming difficult implementation challenges and creating a foundation for future research in this area. While there has been continuing recent work toward improving the accuracy and performance of classification methods, the ability to leverage the cloud will be critical to the success of any real-world sign language recognition system, and this research lays the groundwork for such systems.

REFERENCES

Print Publications:

[Baranwal14]
N. Baranwal, N. Singh, and G. C. Nandi, “Indian Sign Language gesture recognition using Discrete Wavelet Packet Transform”, *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT)*, pp. 573–577, Jan. 2014.

[Belongie02]
S. J. Belongie, J. Malik, and J. Puzicha, “Shape Matching and Object Recognition Using Shape Contexts”, *PAMI*, vol. 24, no. 4, pp. 509–522, 2002.

[Chang10]
R. Chang, F. Wang, and P. You, “A Survey on the Development of Multi-touch Technology”, *2010 Asia-Pacific Conference on Wearable Computing Systems (APWCS)*, pp. 363–366, Jan. 2010.

[Chang11]
Y.-S. Chang, S.-H. Hung, N. J. C. Wang, and B.-S. Lin, “CSR: A Cloud-Assisted Speech Recognition Service for Personal Mobile Device”, *ICPP ’11: Proceedings of the 2011 International Conference on Parallel Processing*, Sep. 2011.

[Chuan14]
C.-H. Chuan, E. Regina, and C. Guardino, “American Sign Language Recognition Using Leap Motion Sensor”, *ICMLA*, pp. 541–544, Jan. 2014.

[Cuervo10]
E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: making smartphones last longer with code offload”, *MobiSys*, pp. 49–62, Jan. 2010.

[Elakkiya14]
R. Elakkiya, K. Selvamani, and S. Kanimozhi, “A framework for recognizing and segmenting sign language gestures from continuous video sequence using boosted learning algorithm”, *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pp. 498–503, Jan. 2014.

[Furht10]
B. Furht and A. Escalante, “Handbook of Cloud Computing, 1st edition”, Sep. 2010.

[Kumarage11]
D. Kumarage, S. Fernando, P. Fernando, D. Madushanka, and R. Samarasinghe, "Real-time sign language gesture recognition using still-image comparison & motion recognition", *Industrial and Information Systems (ICIIS), 2011 6th IEEE International Conference on*, pp. 169–174, Jan. 2011.

[Lichtenauer07]
J. F. Lichtenauer, G. A. ten Holt, E. A. Hendriks, and M. J. T. Reinders, "Sign language detection using 3D visual cues", *AVSS '07: Proceedings of the 2007 IEEE Conference on Advanced Video and Signal Based Surveillance*, pp. 435-440, Sep. 2007.

[Lu97]
S. Lu, S. Igi, H. Matsuo, and Y. Nagashima, "Towards a Dialogue System Based on Recognition and Synthesis of Japanese Sign Language", *Proceedings of the International Gesture Workshop on Gesture and Sign Language in Human-Computer Interaction*, Sep. 1997.

[Min07]
S. Min, S. Oh, G. Kim, T. Yoon, C. Lim, Y. Lee, and K. Jung, "Simple glove-based Korean finger spelling recognition system", *ICCSA'07: Proceedings of the 2007 International Conference on Computational Science and its Applications*, Volume Part I, pp. 1063-1073, Aug. 2007.

[Nicholson08]
A. J. Nicholson and B. D. Noble, "BreadCrumbs: forecasting mobile connectivity", *MOBICOM*, pp. 46–57, Jan. 2008.

[Ong05]
S. C. W. Ong and S. Ranganath, "Automatic sign language analysis: a survey and the future beyond lexical meaning", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 873–891, Jan. 2005.

[Perlmutter92]
D. M. Perlmutter, "Sonority and syllable structure in American Sign Language", *Linguistic Inquiry*, vol. 23, no. 3, pp. 407-442, Summer 1992.

[Phadtare12]
L. K. Phadtare, R. S. Kushalnagar, and N. D. Cahill, "Detecting hand-palm orientation and hand shapes for sign language gesture recognition using 3D images", *Image Processing Workshop (WNYIPW), 2012 Western New York*, pp. 29–32, Jan. 2012.

[Rose01]
R. C. Rose, S. Parthasarathy, B. Gajic, A. Rosenberg, and S. Narayanan, “On the implementation of ASR algorithms for hand-held wireless mobile devices”, *Acoustics, Speech, and Signal Processing, 2001. Proceedings ICASSP ’01. 2001 IEEE International Conference on*, vol. 1, pp. 17–20, Jan. 2001.

[Rose06]
R. C. Rose and I. Arizmendi, “Efficient client–server based implementations of mobile speech recognition services”, *Speech Communication*, vol. 48, no. 11, pp. 1573–1589, Nov. 2006.

[Shanableh07]
T. Shanableh and K. Assaleh, “Arabic sign language recognition in user-independent mode”, *International Conference on Intelligent and Advanced Systems*, pp. 597–600, Jan. 2007.

[Swee07]
T. T. Swee, S. H. Salleh, A. K. Ariff, C.-M. Ting, S. K. Seng, and L. S. Huat, “Malay Sign Language Gesture Recognition system”, *International Conference on Intelligent and Advanced Systems*, pp. 982–985, Jan. 2007.

[Veitch11]
R. Veitch, R. Woods, and L. M. Aubert, “GPU acceleration of automated speech recognition for mobile devices”, *Industrial Informatics (INDIN)*, pp. 823–828, Jan. 2011.

[Wang05]
C. Wang, X. Chen, and W. Gao, “A comparison between etymon- and word-based Chinese sign language recognition systems”, *GW'05: Proceedings of the 6th international conference on Gesture in Human-Computer Interaction and Simulation*, pp. 84-87, May 2005.

[Zaykovsky08]
D. Zaykovsky, “Survey of the Speech Recognition Techniques for Mobile Devices”, *International Journal of Speech Technology*, vol. 11, no. 2, pp. 63–72, Jun. 2008.

Electronic Sources:

[Fundamentals18]
Fundamentals of Garbage Collection | Microsoft Docs: 2018. *https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals*. Accessed: 2018-03-26.

[JamesNK18]
"JamesNK/Newtonsoft.Json: Json.NET is a popular high-performance JSON framework for .NET": 2018: *https://github.com/JamesNK/Newtonsoft.Json*. Accessed: 2018-03-26.

[Json17]
RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format: 2017. *https://tools.ietf.org/html/rfc8259*. Accessed: 2018-03-26.

[mgravell18]
"mgravell/protobuf-net: Protocol Buffers library for idiomatic .NET": 2018. *https://github.com/mgravell/protobuf-net*. Accessed: 2018-03-26.

[Mirashe10]
S. P. Mirashe and N. V. Kalyankar, "Cloud Computing", 01-Jan-2010. [Online]. Available: http://arxiv.org/abs/1003.4074.

[Performance17]
Performance Counters in the .NET Framework | Microsoft Docs: 2017. *https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/performance-counters*. Accessed: 2018-03-26.

[Protobuf18]
Protocol Buffers | Google Developers: 2018. *https://developers.google.com/protocol-buffers/*. Accessed: 2018-03-26.

[WebApi18]
"WebApiContrib/WebAPIContrib.Core: Community Contributions for ASP.NET Core": 2018. *https://github.com/WebApiContrib/WebAPIContrib.Core*. Accessed: 2018-03-26.

APPENDIX A

AMERICAN SIGN LANGUAGE GESTURES

COMMON

- Hello
- No
- Please
- Thank You
- Yes

PEOPLE

- Man
- Woman

SCHOOL

- School
- Student
- Teacher

HOME

- Bathroom
- Bicycle
- House

FAMILY

- Boyfriend
- Father
- Girlfriend
- Husband
- Mother
- Wife

TIME

- Morning
- Night

ANIMALS

- Cat
- Dog

DESCRIPTIONS

- Cold
- Happy
- Hot

APPENDIX B

IRB APPROVAL

UNF
UNIVERSITY of NORTH FLORIDA.

Office of Research and Sponsored Programs
1 UNF Drive
Jacksonville, FL 32224-2665
904-620-2455 FAX 904-620-2457
Equal Opportunity/Equal Access/Affirmative Action Institution

MEMORANDUM

DATE: September 22, 2017

TO: Ching-Hua Chuan, Ph.D.
Computing

UNF IRB Number: 967107-3
Approval Date: 09-22-2017
Expiration Date: 09-22-2018
Processed on behalf of UNF's IRB

FROM: Dr. Jennifer Wesely, Chairperson
On behalf of the UNF Institutional Review Board

RE: Review of Revisions for New Project by the UNF Institutional Review Board
IRB#967107-3: "American Sign Language Gesture Research"

This is to advise you that your project titled "American Sign Language Gesture Research" underwent "Expedited" Categories 6 & 7 review on behalf of the UNF Institutional Review Board. Your reviewer recommended approval without further modifications. As of today, your proposal which required modifications was approved.

This approval applies to your project in the form and content as submitted to the IRB for review. All participants must receive a stamped and dated copy of the approved informed consent document when possible. Any variations or modifications to the approved procedures or documents must be cleared with the IRB prior to implementing such changes. *For example*, if you plan to make changes to your stamped and dated informed consent form, it will be necessary to submit a copy of the revised form via an amendment so that it can be reviewed and approved prior to use. Once approved, a new stamp and date will be included on the revised consent form so that it can be used. To submit an amendment, please complete an Amendment Request Document and submit it along with any updated documents affected by the changes via a new package in IRBNet. Any unanticipated problems involving risk and any occurrence of serious harm to subjects and others shall be reported by completing this Event Report Form and sending it promptly to the IRB within 3 business days.

Your study has been approved for a period of 12 months as of 09/22/2017. If you would like your project to continue for more than one year, you will be required to provide a completed Status Report and other continuing review documentation to the UNF IRB prior to **08/22/2018.** An extension will be necessary if your study will be continuing past the 1-year anniversary of the approval date. *We ask that you submit your status report and other continuing review information 30 days before the expiration date as noted above to allow time for review and processing.* When you are ready to close your project, please complete a Closing Report Form. Please note that it will be necessary to create a new package in IRBNet in order to submit amendments, status

reports, or closing reports in the future. All applicable records relating to this research shall be retained for at least 3 years after completion of the research.

CITI Training for this Project:

Name	CITI Expiration Date
Dr. Chuan	03/28/2019
Mr. Blair	01/16/2020
Mr. Southard	01/04/2020

CITI Course Completion Reports are valid for 3 years. The CITI training for renewal will become available 90 days before the current CITI training expires. Please renew your CITI training when necessary and ensure that all key personnel maintain current CITI training. Individuals can access CITI by following this link: http://www.citiprogram.org/. Should you have questions regarding your project or any other IRB issues, please contact the research integrity unit of the Office of Research and Sponsored Programs by emailing IRB@unf.edu or calling (904) 620-2455.

This letter has been electronically signed in accordance with all applicable regulations, and a copy is retained within UNF's records. All records shall be accessible for inspection and copying by authorized representatives of the department or agency at reasonable times and in a reasonable manner. A copy of this approval may also be sent to the dean and/or chair of your department.

UNF IRB Number: 967107-3
Approval Date: 09-22-2017
Expiration Date: 09-22-2018
Processed on behalf of UNF's IRB

VITA

James Blair has a Bachelor of Science degree from Wake Forest University in Chemistry, 2010 as well as a Master of Science degree from The University of North Carolina Chapel Hill in Chemistry, 2012. After fulfilling several computing prerequisites, James expects to receive a Master of Science in Computing and Information Systems from the University of North Florida, April 2018. Dr. Ching-Hua Chuan of the University of North Florida is serving as James's thesis advisor. James is currently employed as a lead software engineer at Feature[23] and has been with the company for 5 years.

James has ongoing interests in mobile and distributed systems, and the patterns and practices used in the construction of such systems. James has extensive programming experience in C#, Objective-C, and Swift, as well as familiarity with other languages such as C++, Java, and Kotlin.