EgoSense: A Minimally Intrusive Monitor of Social and Physical Activities

Octav Chipara, Farley Lai, Austin Laugesen, S. Shabih Hasan
Department of Computer Science
University of Iowa

Background

Despite the advance of modern technology, traditional data collection methods such as questionnaires and passive reports are still widely used to keep track of social and physical activities among people. With the data gathered, we can construct a social network that describes the interactions and relationships between people. Based on the formulated network, it is viable to measure an individual's quality of life [4] and even monitor his or her psychological states over time. However, to measure in this way does not scale well when the number of people grows largely. Besides, all the data cannot be collected at an accurate and high temporal resolution because most data collections are done manually with unavoidable labor cost. In view of the limitations, we manage to automate the data collection process so as to facilitate the mining and analysis of social networks. The resulting work is EgoSense, that utilizes advanced modern sensing technology available on mobile devices to monitor social and physical activities of people with minimal intrusion.

The EgoSense project is broken down into several phases. In phase I, we aim at developing a system prototype which is supposed to be deployed in a group home context and focus on constructing an interaction network by tracking the identities of speakers in real-time conversations. To achieve the goal, a speaker identification application is essential and several relevant digital speech processing techniques such as feature extraction and Gaussian mixture models have to be studied. In general, speaker identification can be viewed as an instance of audio surveillance which, in contrast with video surveillance using cameras, exploits microphones to capture speech samples to detect abnormal events from unusual speaking patterns. In such an application, the event detection makes little sense if the speakers can not be identified. Besides, speaker identification is also able to assist memory in life logging applications. For example, current calendar applications may remind you of where and when to do something with somebody but fail to notify you of something to do when you meet someone. However, with the introduction of speaker identification, even personal life logging is made feasible.

On the other hand, mobile devices equipped with accelerometer, proximity sensors and microphones typically have only limited computing power and battery life. Additionally, training data acquisition is also a challenge to speaker identification [1] since it is almost impossible to acquire the speech samples from all an individual's acquaintances in advance. In light of these limitations and challenges, EgoSense assumes a group home context where only a fixed number of

patients are considered and their speech samples can be collected in advance. Further, EgoSense adopts client-server architecture, requiring mobile devices only to record speech and perform features extraction. Computing-intensive tasks such as speech diarization, training and matching speaker models are left for high performance servers. As a consequence, EgoSense addresses the limitations and challenges appropriately.

Next, we briefly go through the procedure of speaker identification. Figure 1 shows a high level block diagram where a mobile device is responsible for capturing speech samples, dividing the samples into frames as the units of feature extraction and transmitting the computed features to a remote server. The remote server performs the speaker identification by first diarizing the features into clusters of feature segments. Here, one cluster corresponds to a particular speaker and the feature segments of a cluster belong to the speech of that speaker. Then the server tries to match each cluster to existing speaker models and compute scores representing the degree of matching. For those clusters with high scores, the matched speaker identities are logged. Some post-processing and analyses may be performed depending on the application. Currently, we only concern the data flow before speaker identification and the accuracy of speaker identification.

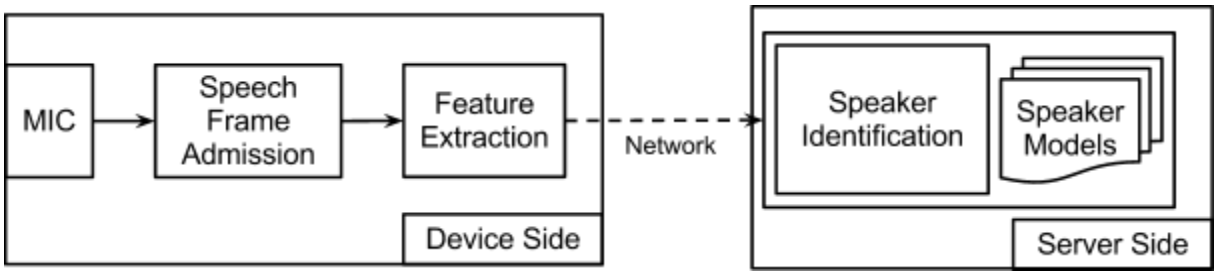

Figure 1. The procedure of speaker identification involves the device side which computes and transmits essential speech features to the server side to match speaker identities.

For the data flow before speaker identification, feature extraction is the most crucial operation on the device side. The feature for speaker identification is the Mel-frequency cepstrum (MFC) which can be represented by a set of Mel-frequency cepstral coefficients (MFCCs). The MFCCs can be computed as follows [3].

1. Derive the Fourier transform of a windowed signal, typically 512 samples
2. Use triangular filters to map the powers of the spectrum obtained above onto the mel scale which is a perceptual scale of pitches judged by listeners to be equal in distance from one another
3. Derive the logs of the powers at each of the mel frequencies
4. Derive the discrete cosine transform of the list of mel log powers, treating it as a signal
5. The MFCCs are the amplitudes of the resulting spectrum

For the accuracy of speaker identification, basic knowledge of Gaussian mixture models is necessary to understand how to train and match a speaker model with a set of speech features. We start with the Gaussian probability density function (pdf) for a d-dimensional random variable x

with μ as the mean vector and Σ as the variance-covariance matrix as follows [10]:

$$g_{(\mu,\Sigma)}(x) = \frac{1}{\sqrt{2\pi}^d \sqrt{det(\Sigma)}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}$$

So-call speech features are examples of those *d*-dimensional variables. Now, given a *d*-dimensional sample point, its *likelihood* regarding a Gaussian model with parameters $\Theta = (\mu, \Sigma)$ is just the value of the Gaussian pdf for that sample point. If a set of independent identically distributed (i.i.d.) sample points are given, the *joint likelihood* of the points is the product of the likelihood for each point. Here, the likelihood can be interpreted as the probability that the point belongs to the Gaussian model. In practice, we compute the *log-likelihood* instead of the simple likelihood to avoid the computation of the exponential in the Gaussian pdf. Besides, the log-likelihoods keep the same relations of order as the likelihoods because *log*(*x*) is a monotonically increasing function.

$$p(X \mid \Theta) = \prod_{i=1}^{N} p(x_i|\Theta) = \prod_{i=1}^{N} g_{(\mu,\Sigma)}(x_i) \Leftrightarrow log\, p(X \mid \Theta) = \sum_{i=1}^{N} log\, p(x_i \mid \Theta) = \sum_{i=1}^{N} log\, g_{(\mu,\Sigma)}(x_i)$$

Next comes the *Bays' decision rule* that determines which model a sample point should belong to with the highest probability. Given a set of classes q_k, characterized by a set of known parameters Θ, a set of one or more feature vectors *X* is said to belong to a particular class if the following rule holds.

$$X \in q_k \text{ if } P(q_k \mid X, \Theta) > P(q_j \mid X, \Theta), \forall j \neq k$$

The *Bays' Law* specifies:

$$P(q_k \mid X, \Theta) = \frac{P(X \mid q_k, \Theta) P(q_k \mid \Theta)}{P(X \mid \Theta)}$$

where $P(X \mid q_k, \Theta)$ is the joint likelihood of the sample vectors *X* with respect to the Gaussian model of the class (q_k, Θ_k), $P(q_k \mid \Theta)$ is the *a-priori* class probability for the class q_k which defines the absolute probability of occurrence for class q_k and $P(X \mid \Theta)$ stands for $\Sigma_j P(X \mid q_j, \Theta) P(q_j \mid \Theta)$. Since $P(X \mid \Theta)$ is independent of different classes, we need only to compute $log(P(X \mid q_k, \Theta) P(q_k \mid \Theta)) = logP(X \mid q_k, \Theta) + logP(q_k \mid \Theta)$ to determine which class the set of feature vectors *X* should belong to.

What follow are the concepts of supervised and unsupervised trainings. In supervised training, each training sample is known a-priori to belong to a particular class while in unsupervised training, we need to classify the training samples into different classes without knowing a-priori which sample belongs to which class. Several clustering algorithms such as K-means, Viterbi-EM and EM (Expectation Maximization) serve for this purpose. Those typically consist of the following components:

1. a set of models q_k, defined by some parameters Θ
2. a measure of membership specifying the goodness of fit of the models to a sample, which could be a distance or a probability
3. a procedure to update the model parameters in function of membership information

We take the popular EM algorithm for example which performs Gaussian clustering. In this algorithm, each data point belongs to each class with a probability. The algorithm is described as follows.

- Initial: start from *K* Gaussian models $\Theta(\mu_k, \Sigma_k)$, $k = 1, 2, ..., K$, with $P(q_k) = 1/K$

- Repeat:
  - Estimation: compute the probability $P(q_k^{(old)} \mid x_n, \Theta^{(old)})$ for each data point x_n to belong to the class $q_k^{(old)}$:
    $P(q_k^{(old)} \mid x_n, \Theta^{(old)}) = \frac{P(x_n \mid q_k^{(old)}, \Theta^{(old)}) P(q_k^{(old)} \mid \Theta^{(old)})}{P(X \mid \Theta)} = \frac{P(x_n \mid q_k^{(old)}, \Theta^{(old)}) P(q_k^{(old)} \mid \mu_k^{(old)}, \Sigma_k^{(old)})}{\sum_j P(x_n \mid q_j^{(old)}, \Theta^{(old)}) P(q_j^{(old)} \mid \mu_j^{(old)}, \Sigma_j^{(old)})}$
  - Maximization:

    update the means $\mu_k^{(new)} = \frac{\sum_{n-1}^{n} x_n P(q_k^{(old)} \mid x_n, \Theta^{(old)})}{\sum_{n-1}^{n} P(q_k^{(old)} \mid x_n, \Theta^{(old)})}$

    update the variances $\Sigma_k^{(new)} = \frac{\sum_{n-1}^{n} P(q_k^{(old)} \mid x_n, \Theta^{(old)})(x_n - \mu_k^{(new)})(x_n - \mu_k^{(new)})^T}{\sum_{n-1}^{n} P(q_k^{(old)} \mid x_n, \Theta^{(old)})}$

    update the priors $P(q_k^{(new)} \mid \Theta^{(new)}) = \frac{1}{N} \sum_{n=1}^{N} P(q_k^{(old)} \mid x_n, \Theta^{(old)})$
- Until: the total likelihood $\Psi(\Theta)$ increase for the training data falls under some desired threshold.

$$\Psi(\Theta) = log P(X \mid \Theta) = log \sum_{k=1}^{K} P(q_k \mid X, \Theta) p(X \mid \Theta)$$

$$\geq \sum_{k=1}^{K} P(q_k \mid X, \Theta) log\, p(X \mid \Theta) = \sum_{k=1}^{K} \sum_{n=1}^{N} P(q_k \mid x_n, \Theta) log\, p(x_n \mid \Theta)$$

Now, we extend the Gaussian probability density to Gaussian mixture density which is a weighted sum of M component densities [2] as follows:

$$p(X \mid \lambda) = \sum_{i=1}^{M} p_i g_i(X)$$

where X is a set of d-dimensional random vectors, $g_i(X)$, $i = 1, 2, ..., M$ are the Gaussian component densities as mentioned above and p_i, $i = 1, ..., M$ are the mixture weights. For speaker identification, each speaker is represented by a Gaussian mixture model (GMM) with a corresponding $\lambda = \{p_i, \mu_i, \Sigma_i\}$, $i = 1, 2, ..., M$. Finally, the idea of training a GMM for a speaker is basically to estimate the λ which best matches the distribution of a given training set of speech feature vectors. The previously introduced EM algorithm applies here.

Approach

Based our previously developed EgoSense toolkit, a system is typically modeled as a pipeline of connected components according to the data flow. The connection between components can be local or over a network. In the EgoSense application, we aim to design a system with minimal thread and constant memory allocation. However, for some components such as the audio capturer running in a separate thread dispatched by the Android system [9], we need a queue component to serve as an intermediate buffer to facilitate synchronized message exchange with another component running in the main thread. Hence, the two components connected through a queue

work close to the producer-consumer pattern. This is where additional threads come in. For memory allocation, the components are classified into sources, filters and sinks. Only source components need to pre-allocate message buffers to fill data and create a message pool to manage the message buffers. Filter and sink components merely forward or receive message buffers from upstream components, incurring no memory allocation. On the other hand, the source and sink components are responsible for recycling the message buffers from previous components. A tap component is attached to each source and sink component to serve this purpose. As long as each message buffer is eventually recycled in a tap component, there is no worry about memory leak and performance decline due to frequent memory allocations.

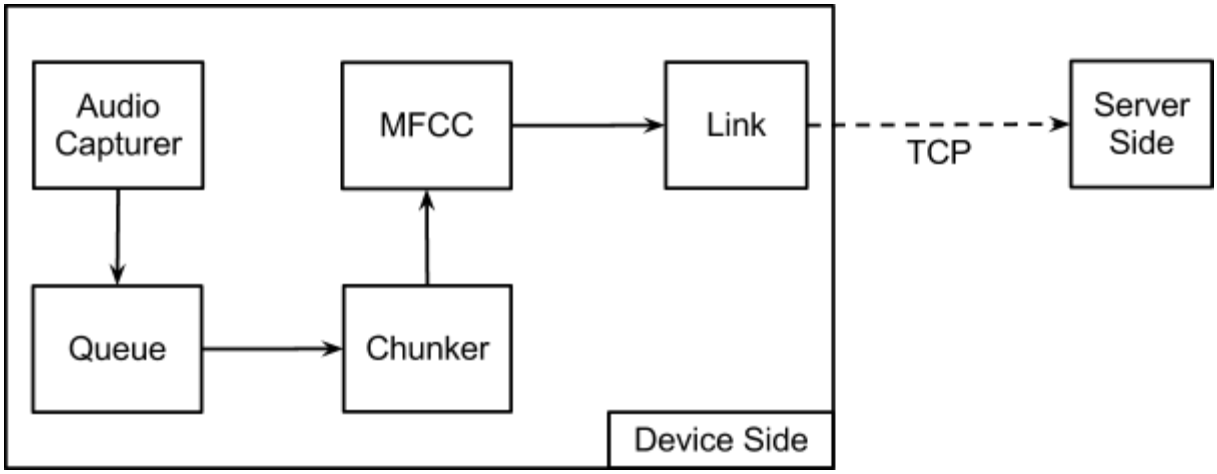

Figure 2. The device side block diagram consists of several data processing components connected in a pipeline. At the end of the pipeline, the data is transmitted over a TCP link to a remote server.

Next, we are going to introduce the system block diagrams that describe the process of generating MFCCs from the audio captured on Android devices and transmitting the data to the server side for advanced speech processing such as speaker identification. To simplify the system block diagrams, we omit those tap components. As is shown in Figure 2, on the device side, audio samples are captured from a microphone and then pushed into a queue component waiting for the main thread running the chunker component to pull. Subsequently, the chunker divides the audio samples into chunks with a multiple of 512 samples required by the MFCC component because of the fast fourier transform (FFT) used to extract MFCCs. Since the audio samples generated from the microphone might not be a multiple of 512 samples, the chunker has to deal with the remaining samples which should be packed next time. The MFCC component acts like a filter converting the samples into MFCCs. To be specific, every 512 audio samples are converted into 32 float MFCCs with double precision. Finally, the produced MFCCs are packed into a multiple of 32 float numbers and fed into the link component. The link component serves as a TCP client connecting to a remote server to transmit the MFCCs. To sum up, this is how the device side works.

Though the scenario on the device side is conceptually simple, there are still performance

concerns with memory allocation and copying. In this case, the audio capturer, chunker and MFCC are all the source components because those are either the first component that generates data or produces messages with different buffer size after processing the input data. Since all the necessary message buffers are pre-allocated in system initialization, there should be little runtime overhead of memory allocation. Another concern is raised by the MFCC component which converts the audio samples into MFCCs using native code generated by the MATLAB Coder [8]. This may incur memory copying overhead because typically an Android application runs in a virtual machine where the heap memory is managed by a garbage collector but the memory including the stack and heap used by the native code is not managed in case the data passed in is garbage collected. Therefore, the data needs to be copied back and forth between the managed heap and the unmanaged device memory. In the long run, the repeated memory copying not only harms the system performance but also wastes more energy. To address this issue, we decide to allocate message buffers from the unmanaged memory. Such buffers are called direct buffers in the Java Native Interface (JNI) terminology [6] which prevent the virtual machine from unnecessary copying. The Java ByteBuffer introduced in the new I/O package provides this mechanism. Nonetheless, the interfaces generated by the MATLAB Coder need some adaptations to cooperate with direct buffers. More technical details will be given in the next implementation section.

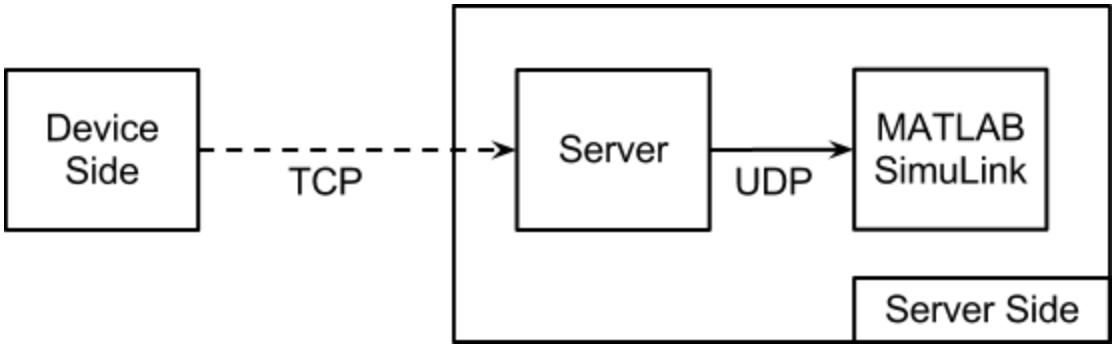

Figure 3. The server side block diagram consists of a server component and a MATLAB component. The server component forwards the incoming data from a device to the MATLAB SimuLink model that performs speaker identification.

Figure 3 demonstrates the server side block diagram where a server component accepts a TCP connection from a device which transmits MFCCs generated from real-time captured audio. After receiving the data from a device, the server component then forwards the data in UDP datagrams to a MATLAB SimuLink model [7] which provides advanced speech processing of the input. The UDP datagram should contains a multiple of 32 float MFCCs. In this report, we just focus on how to extract features on the device side efficiently and verifying the MFCCs received on the server side. The next phase of our project will concentrate on the MATLAB SimuLink model.

Implementation

We choose the Android platform to implement our device side application since it is open and

popular, covering different market segments. Our application is designed to adapt to devices of different capabilities in audio sampling, sensing and performance so that the ultimate solution caters to diverse budgets. For example, instead of using a single powerful smartphone to collect all the required data, our application allows multiple devices collaborating to gather various data, some of which need not to be high-end. We expect our solution to make best effort for general use.

Next, we are going to address the issue of passing message buffers to native code without extra memory copying. Typically, an Android application is able to invokes native code via the JNI which can be written manually or generated using the SWIG compiler [5]. The native function interface generated by the MATLAB Coder to compute the MFCCs is given as follows.

a_melcepst.h

```
void a_melcepst(const real_T s[512], real_T fs, int32_T nc, emxArray_real_T *c);
```

The parameter `s[512]` indicates the function accepts 512 float audio samples at one time and the resulting MFCCs will be stored in the structure of type `emxArray_real_T` where there is a pointer to a dynamically allocated buffer as output. Our idea is to pass a pointer to the input message buffer as `s[512]` and another pointer to the output message buffer that replaces the dynamically allocated output buffer in the structure of type `emxArray_real_T`. Since an input message of the MFCC component contains a multiple of 512 audio samples and each time only 32 float MFCCs are computed, we need additional parameters to specify the offsets of the input and output message buffers. Consequently, we wrap the interface a_melcepst() with the following interface which accepts extra parameters `r[32]` as the output message buffer, `offset1` as the offset of the input message buffer and `offset2` as the offset of the output message buffer. Obviously, the wrapper adjusts the offsets of the message buffers each time before calling a_melcepst() so that the same message buffers can be reused across several invocations.

a_melcepst_wrap.c

```
void a_melcepst_wrap(   const real_T s[512], int32_T offset1,
                        real_T fs, int32_T nc,
                        emxArray_real_T *c, real_T r[32], int32_T offset2) {
      s = &s[offset1];
      r = &r[offset2];
      if(c->data != r) {
            if(c->allocatedSize == 0) free(c->data);
            c->size[0] = 1;
            c->size[1] = 32;
            c->data = r;
            c->allocatedSize = 64;
            c->canFreeData = FALSE;
      }
      a_melcepst(s, fs, nc, c);
}
```

Last but not least, we go through the essential code segment of the SWIG interface file where

the memory address of the message buffer is extracted from a Java ByteBuffer object so as to be passed to the native code. In a SWIG interface file, a special directive `typemap` is used to match a particular parameter data type in the native interfaces and specify the operations to convert the corresponding Java data type. Here, the key method **`GetDirectBufferAddress()`** extracts the memory address of the message buffer which is a ByteBuffer object. In this way, whenever a native interface with a parameter `s[512]` of type `const real_T` gets called, the address extracted from a ByteBuffer object passed from the binding interface will be passed as the corresponding argument to the native interface. Likewise, the parameter `r[32]` of type `real_T` corresponds to the address of the output message buffer which is also a ByteBuffer object. As a result, the input and output message buffers both can be passed to the native interface for read/write operations without incurring memory copying overhead.

melcepst.i

```
%typemap(in) (const real_T s[512]) {
  $1 = (real_T*)(*jenv)->GetDirectBufferAddress(jenv, $input);
}

%typemap(jni) (const real_T s[512]) "jobject"
%typemap(jtype) (const real_T s[512]) "java.nio.ByteBuffer"
%typemap(jstype) (const real_T s[512]) "java.nio.ByteBuffer"
%typemap(javain) (const real_T s[512]) "$javainput"
%typemap(javaout) (const real_T s[512]) {
  return $jnicall;
}

%typemap(in) (real_T r[32]) {
  $1 = (real_T*)(*jenv)->GetDirectBufferAddress(jenv, $input);
}

%typemap(jni) (real_T r[32]) "jobject"
%typemap(jtype) (real_T r[32]) "java.nio.ByteBuffer"
%typemap(jstype) (real_T r[32]) "java.nio.ByteBuffer"
%typemap(javain) (real_T r[32]) "$javainput"
%typemap(javaout) (real_T r[32]) {
  return $jnicall;
}
```

Experiment

In the phase of our project, the most crucial concern is the correctness of the feature extraction. To verify the correctness of the MFCCs computed on the device side, we prepared for a raw PCM audio recording beforehand as the input samples. The audio sample format is 16-bit little-endian and the sampling rate is 16000. There are 4096 blocks of 512 samples in the audio recording, namely, 4MB in total. The resulting MFCCs should be 4096 blocks of 32 64-bit float MFCCs, viz. 1MB in total. Figure 3 illustrates the test scenario on the device side and the server side. Compared

with Figure 2, the audio capturer is replaced with a From Disk component which supports reading from a file. On the server side, the server component forwards the received data to the To Disk component which supports writing to a file in local storage. As is shown, the data flow can be easily altered by replacing only a few components. Most other components remain unchanged for reuse. After the MFCCs computed on the device side are saved to a file on the server side, the file is compared with the results generated from the same native code compiled on the server side. We use the `diff` command to compare the files.

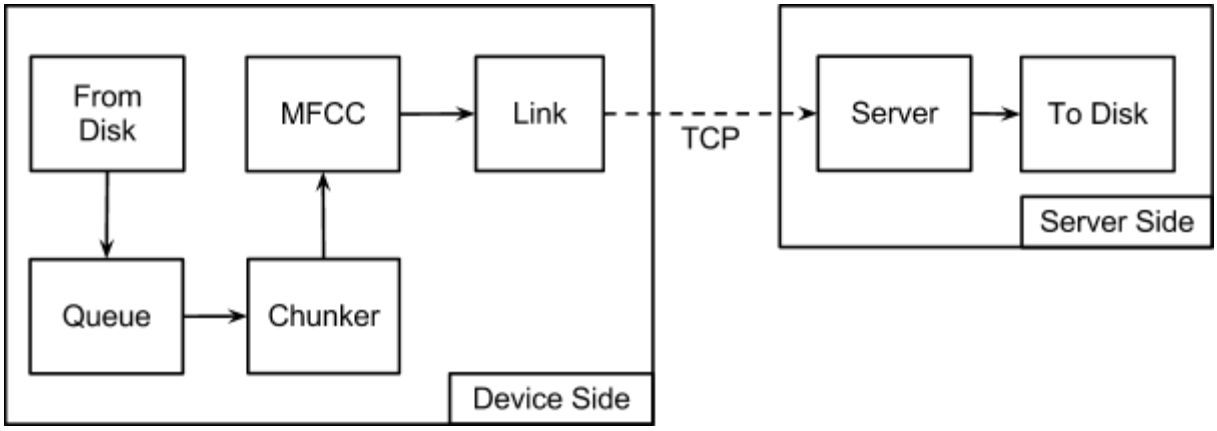

Figure 3. In this scenario, audio samples are read from a raw PCM file in local storage on the device side while the computed MFCCs are saved to a file on the server side for comparison.

Discussion and Conclusions

As expected, the MFCCs computed on both device and PC sides are exactly the same, proving the correctness of the system design. However, there is one thing worthy of notice. It is the endianness setting of the MATLAB Coder. Though a ByteBuffer object can be configured as big-endian or little-endian, we find that only when the MATLAB Coder sets the little-endian option does the generated native code produce the expected results in the little-endian ByteBuffer object. After further investigation, we jump to the conclusion that most Android devices are configured to be little-endian by default even though the CPU is bi-endian. Thus, we are confident that endianness will not be an issue with the distribution of our application on other devices.

In summary, we give an overview of EgoSense as well as essential theoretical backgrounds in feature extraction and the Gaussian mixture model. The proposed approach is based on our previously developed toolkit, with which the system on both device side and server side can be viewed as pipelines following the data flow. Besides, we address the memory management issues such as how to minimize thread and memory allocation. We also prevent unnecessary memory copying by passing direct buffers to native code. Finally, we conduct an experiment to verify the MFCCs computed. Everything works like a charm except endianness. Fortunately, endianness is not an issue as long as little endianness is configured by default. In the future, we plan to integrate a speech detection component to filter out non-speech audio samples and conduct several experiments to tune a set of effective parameters relevant to the speaker identification. Moreover,

devices of different capabilities will also be taken into consideration.

References

[1] L. Hong, A. J. B. Brush , B. Priyantha , A. K. Karlson and J. Liu, “SpeakerSense: energy efficient unobtrusive speaker identification on mobile phones,” Proceedings of the 9th international conference on Pervasive computing, June 12-15, 2011, San Francisco, USA

[2] D.A. Reynolds and R.C. Rose, "Robust text-independent speaker identification using Gaussian mixture speaker models," Speech and Audio Processing, IEEE Transactions on , vol.3, no.1, pp.72-83, Jan 1995, doi: 10.1109/89.365379

[3] X. Huang, A. Acero and H-W. Hon, “Spoken Language Processing: A Guide to Theory, Algorithm and System Development,” Prentice Hall PTR, NJ, 2001, pages 273-329.

[4] J. E. Ware, S. D. Keller, B. Gandek, J. E. Brazier, M. Sullivan amd the IQOLA Project Group, “Evaluating Translations of Health Status Questionnaires: Methods From the IQOLA Project,” International Journal of Technology Assessment in Health Care, vol. 11, no. 3, p. 525, Mar. 2009.

[5] SWIG: Simplified Wrapper and Interface Generator, http://www.swig.org

[6] JNI Enhancements, http://docs.oracle.com/javase/1.4.2/docs/guide/jni/jni-14.html

[7] MATLAB SimuLink, http://www.mathworks.com/products/simulink/

[8] MATLAB Coder, http://www.mathworks.com/products/matlab-coder/

[9] Android, a open-source mobile platform, http://www.android.com/

[10] Introduction to Gaussian Statistics and Statistical Pattern Recognition, http://scgwww.epfl.ch/matlab/student_labs/2005-2006/labs/Lab06_Introduction_GMM/labo9_files/labman_English.pdf