
UNIVERSITÀ DI PISA

Industrial Applications

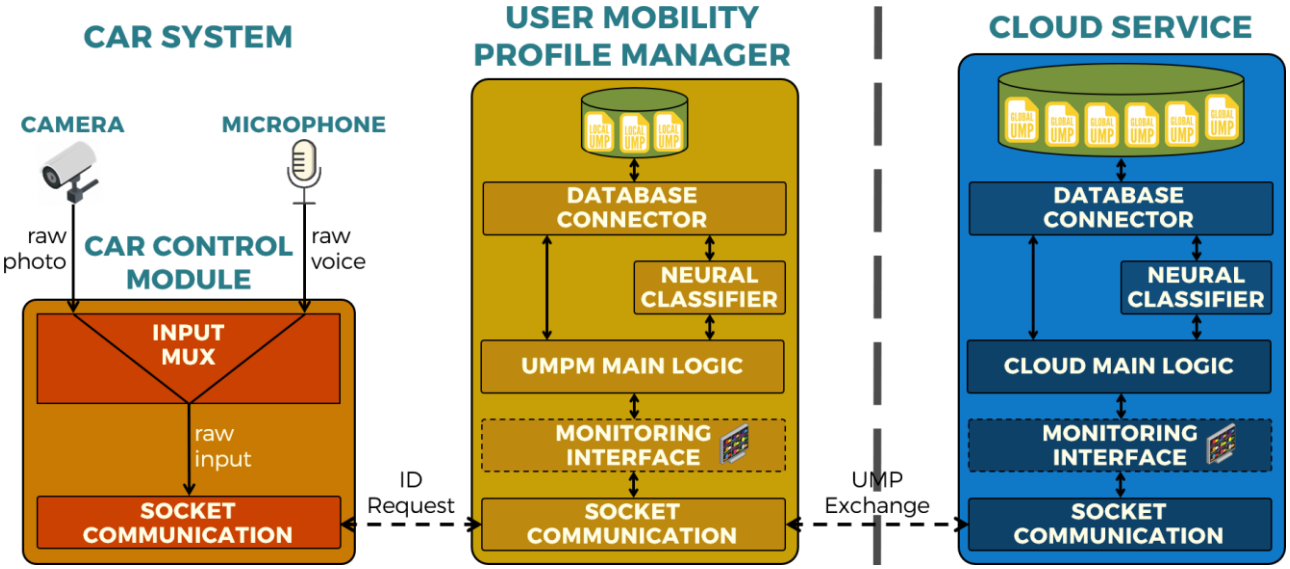
# The User Mobility Profile Prototype Report

## Table of Contents

Prototype Architecture ..... 1
Hardware Deployment ..... 2
Implementation Details ..... 3
Library Details ..... 3
  Facial Recognition ..... 3
  Voice Recognition ..... 4
Prototype Benchmarks ..... 5
Concluding Remarks ..... 5
  Facial Recognition System ..... 5
  Voice Recognition System ..... 7
  Details and Testing ..... 8

# Prototype Architecture

The following represents the architecture of the User Mobility Profile (UMP) system prototype that was developed for demonstration purposes, where a more detailed description of the functionalities offered by the overall system and the individual components can be found in the related project specification.




The system is composed of three main modules, interchanging data through socket communication

- A Car Control module, representing a stub of the car operating system and control logic, whose task in the context of the prototype is limited to continuously collecting video and voice samples through a camera and microphone and forwarding them to the local User Mobility Profile Manager (UMPM) module to identify the users associated with such inputs.

- A User Mobility Profile Manager (UMPM) module, again deployed in the local car environment and which implements the following functionalities:
    - Each input sample received from the Car Control module is matched against the corresponding biometric feature contained in every UMP stored in a local database, where:
        - If a match was found, a local unique identifier associated with such user is returned to the Car Control module.
        - If a match was not found, a temporary profile associated with such user is created and its associated unique identifier is returned to the Car Control module, while at the same time a UMP request containing the unmatched feature is forwarded towards the cloud service to retrieve the actual user profile and then merge it with the temporary profile once it has been returned by the cloud service.
    - An interface allowing other software components to read and update the information contained in the UMPs stored in the local database.
      It should be noted that, while in the broader context of the system such interface would be exploited by the applications running in the car ecosystem, within the scope of this prototype it is instead used by an ad-hoc graphical user interface allowing to test the module and monitor its status.
    - The forwarding of each update of the UMP contents to the cloud service for synchronization purposes.

- A mechanism implementing the temporary caching of the UMPs of the users that have not recently been matched by the system, where every time a user is matched, if their UMP is found in the local cache their local unique identifier is immediately returned to the Car Control module, while a message asking for possible updated versions of the associated UMP is forwarded towards the cloud service.

- A Cloud Service module, implementing the following functionalities:
  - Every time a UMP request is received from the UMPM module, the biometric input it contains is matched against the corresponding biometric feature held in every UMP stored in a local database, to subsequently return the associated UMP if a match is found.
  - The mirroring of each UMP content update received from the UMPM module to the corresponding UMP in the local database.
  - The verification that a cached UMP received from the UMPM module corresponds to the latest version of such UMP, returning the updated version if that is not the case.
  - A graphical user interface, that similarly to the one integrated within the UMPM makes it possible to monitor the operations being performed by the module as well as read and update the information of the UMPs stored in the local database.

## Hardware Deployment

With reference to the architecture previously described, the software modules composing the system were deployed as follows:


- Both the Car Control and the User Mobility Profile Manager modules were deployed in a Raspberry Pi 3 B+ equipped with a microphone and a Pi Camera and running the Raspbian operating system.

- The Cloud Service module was deployed in a support notebook running the Mac OS X operating system.

- Network connectivity between the two devices was obtained through the use of an Ethernet cable.

# Implementation Details

Presented below are some details regarding the software implementation of the modules previously described, where more in-depth information can be found in the code source files associated with this report.

- All modules were implemented using the Python language v 3.6, where the latest version (v 3.8) was discarded due to compatibility issues with the Raspbian operating system.
- MongoDB was selected as the Database Management System for storing the User Mobility Profiles, both in the UMPM and in the Cloud Service modules.
- The Cloud Service Database was populated with a collection of photos and voice samples of the developers.
- The speech and face recognition routines were implemented using the AcustID and DLib libraries respectively, while the monitoring interface was implemented with the support of the TkInter library.

# Library Details

## Facial Recognition

We selected the DLib library because is a simple Python library recognize and manipulate faces in a simple way. Built using dlib's state-of-the-art face recognition built with deep learning. The model has an accuracy of 80% on the Labeled Faces in the Wild benchmark. The treatment of the image file it's with the following steps:

1. Upload the face file
2. Extraction of the face features

```
encodings = face_recognition.face_encodings(image, locations)
```

3. Repeat the point 1 and 2 with the local face file. The result going to be the local encode.

```
local_encode = face_recognition.face_encodings(image, locations)
```

4. The `face_locations` function is used to identify the position of faces within the image (our basic working hypothesis is that there is exactly one face in each image), The model used, as specified in the package, is HOG (Histogram of Oriented Gradients), a feature descriptor that uses a linear SVM to do face detection.

```
locations = face_recognition.face_locations(image, model='hog')
```

5. Assessment of the similarity between the two faces

```
results = face_recognition.compare_faces([know_face], face_encoding, 0.6)
```

6. Comparison between each acquired image and the list of images already cataloged using the `compare_faces` function. The `TOLERANCE` parameter has been set to `0.6` because it is an intermediate threshold that takes into account the fallacy of the reference system used. You can do that with the --tolerance parameter.

```
TOLERANCE = 0.6
```

## Voice Recognition

Acoustic fingerprinting is a technique for identifying songs from the way they "sound" rather from their existing metadata. That means that beets' auto tagger can theoretically use fingerprinting to tag files that don't have any ID3 information at all (or have completely incorrect data). This plugin uses an open-source fingerprinting technology called Chromaprint and its associated Web service, called Acoustid. The library code is native on C, but all of the beets core is written in pure Python.

From a higher level, all audio fingerprinting algorithms go through two transformation steps: lossy compression and hashing. We want to preserve as much relevant information as possible with the smallest footprint. The treatment of the audio file it's with the following steps:

1. Upload the audio file
2. Extraction of the duration of the audio file and its fingerprint

```
duration, fp_encoded = acoustid.fingerprint_file(song)
```

3. Decode the fingerprint

```
fingerprint, version = chromaprint.decode_fingerprint(fp_encoded)
```

4. Repeat the point 1 and 2 with the local audio. The result going to be the local fingerprint `fingerprint_local`
5. Assessment of the similarity between the two fingerprints

```
 similarity = fuzz.ratio(fingerprint1, fingerprint_local)
```

# Prototype Benchmarks

The following represents a selection of benchmarks relative to the system execution on the chosen platforms, where it should be noted that the Raspberry Pi is equipped with a Cortex-A53 CPU @1.4GHz with 1GB of LPDDR2 memory, while the supporting notebook is powered by an Intel i9-9880H @4.8 GHz with 16GB of DDR4 memory.

| | Raspberry Pi 3 B+ | Support Notebook |
|---|---|---|
| **Face Recognition Average Matching Time** | 5,54s | 1,20s |
| **Speech Recognition Average Matching Time** | 8,5s | 1,94s |
| **User Identification Average Latency** | 11,25s | |
| **User Identification Latency Variance** | 7,57 | |
| **Average CPU Utilization during matching** | 30% (4 cores / 4 threads) | 2% (8 cores / 16 threads) |
| **Average Memory Utilization during matching** | 412MB | 637MB |

# Concluding Remarks

The system has demonstrated different performances in recognizing a user through facial and voice data. Specifically, with regard to facial data, the user is recognized with a satisfactory accuracy and introducing a latency in line with what could be expected from the use of a neural network such as that provided by DLib; with regard to voice data, the system is not suitable to recognize the user with precision, despite the great computational effort of the AcustID library, the accuracy in the recognition of the audio signature is not usable for the application in use.

## Facial Recognition System

First of all, let's remember that the facial recognition system works in order to extract the essential biometric components concerning the face of a subject from two photos, then the system takes care of producing a matching of the two components.

The neural network that the system employs, substantially plays a role of feature extraction on the photo in analysis, moreover it is necessary that the system, every time that it wants to carry out the matching of a photo, extracts all the images saved in the database and continues to process them through the neural network until it finds one that makes a match.

This level of complexity could be simply avoided by extracting only once the features corresponding to the photos stored in the database, so as to avoid having to use the neural network every time you have to make a matching on all the photos in the database.

This simplification would impose that the system always uses the same neural network (both cloud and vehicle side) and that this network is immutable both in time (every time we want to modify it it will be necessary to update all the features extracted from the images) and in space (i.e. different applications that want to implement the UMP cannot use different neural networks because the application would impose the use of a standardized and not customizable network).

In addition, what appears to be a big problem of inefficiency from the point of view of complexity, is not so from the practical point of view, in fact Raspberry can extract the features in question in an average time of 0.0003275s while what is particularly burdensome for the system is the matching of features that takes 1.32834s.

| | Features Extraction Time | Matching Time |
|---|---|---|
| **Photo-1** | 1,36836s | 0,00046s |
| **Photo-2** | 1,31989s | 0,00028s |
| **Photo-3** | 1,31255s | 0,00027s |
| **Photo-4** | 1,31257s | 0,00030s |
| **Variance** | 0,00077 | 9,025 * 10^-9 |
| **Average** | 1,32834s | 0,0003275s |

In order to take into account both of the above issues, in the prototype it was decided to perform the extraction of all the features at system start-up. In this way we will not have problems of static nature of the networks in time and space and we will not introduce excessive inefficiency measures from the logical point of view.

## Open Issues

This solution introduces in every case the problem to keep updated the features extracted in phase of start-up and the images in the database. To do this it would have been necessary to insert an additional level of consistency between the data. The prototype doesn't have the functionality to register new users, so this problem was left open.

Another problem results from the necessity to make the matching every time of all the features with those just extracted from the input photo, this step is very expensive because as already mentioned the single matching has a computational cost of 1,32834s.

A possible solution to this problem could be to train a neural network ad-hoc on the features to classify, in order to have a rapid and precise tool that performs what has been said above in a single iteration.

The problematic introduced from this system however results in first place that it would need a sufficiently great dataset of extracted features and that besides this net constructed ad-hoc adapts in the time (on-line training) to the insertion of new users in the system (both when the vehicle, or in our case the Raspberry, is active that when it is not). Moreover, it would be very complex to change or update the network in question (changing for example the number of epochs or hyper-parameters to obtain greater accuracy) because it would mean every time re-training it on the whole dataset.

This problem was therefore insurmountable for our project and was left open, as we do not have a dataset of extracted features (which must be specific to the application) and also the cards on which we work are not suitable for training neural networks.

Moreover, in order to improve the performance, we tried to use the multicore calls provided by the library, but despite an increase in CPU usage the performance did not improve, in fact the producer of the library make available this option but he does not sponsor it.

## Voice Recognition System

The performances for the task in analysis of the library AcustID have turned out very inefficient and not to the height to be used operationally inside the prototype (nevertheless it is possible to enable them inside the system if you want).

In order to clarify this result that can appear unexpected it is necessary to make some considerations.

First of all, the AcustID library is responsible for the match between two vocal tracks, recognizing in practice if these two tracks are overlapping and with what tolerance this overlap is possible. In fact, when we tested the application by playing exactly the same vocal source used to make the match, the system correctly recognized the track in question.

What we investigated, however, was whether it was possible to exploit this system of matching two voice tracks, to perform the recognition of the person who was speaking.

This level of abstraction, which we tried to implement by lowering various levels of the matching threshold, does not seem to be implementable through this library, which in fact has been designed (as all audio features of this type in the state of the art) to perform the recognition of similar vocal tracks and not to recognize the user who speaks.

The library was also tested with again unusable performance to recognize a word said in particular by a person ( "hello" ), similar to what happens in voice recognition systems such as Amazon's Alexa, but was not able to perform this task as well.

The problem that we left open then, is based on finding libraries similar to those used by Google or Amazon that are able to carry on the recognition of keyword phrases spoken by the same user to recognize him.

## Details and Testing

The recognition system works in parallel, in other words, face recognition and audio recognition are carried out independently and in parallel. We therefore do not have a fusion of the recognized biometric parameters but, they are kept separate and recognized independently from each other.

Next, we show some empirical results as a confusion matrix, showing true positive (TP), true negative (TN), false positive (FP), false negative (FN). We carried out other tests outside of those visible in person, by framing the phone screen, due to the impossibility of being physically found.

We performed 20 tests both for face and speech recognition both considering the *cloud offline* and *cloud online*, even if the results between the two scenarios do not vary.

The resulting confusion matrices are outlined below:

### Face Recognition

| N = 20 | **Actual Positives** | **Actual Negative** |
|---|---|---|
| **Predicted Positives** | TP = 11 | FP = 1 |
| **Predicted Negatives** | FN = 1 | TN = 7 |

Accuracy = 90 % (TP + TN / N)

Error-rate = 10 % (1 – Accuracy)

Precision = 0.916 % (TP / TP + FP)

Recall = 0.916 % (TP / TP + FN)

### Speech Recognition

| N = 20 | **Actual Positives** | **Actual Negative** |
|---|---|---|
| **Predicted Positives** | TP = 2 | FP = 10 |
| **Predicted Negatives** | FN = 6 | TN = 2 |

Accuracy = 20 % (TP + TN / N)

Error-rate = 80 % (1 – Accuracy)

Precision = 0.16 % (TP / TP + FP)

Recall = 0.25 % (TP / TP + FN)