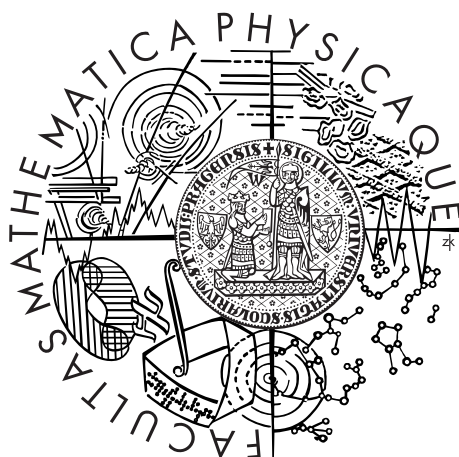Charles University in Prague

Faculty of Mathematics and Physics

# MASTER THESIS

Ondřej Klejch

# Development of a cloud platform for automatic speech recognition

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Mgr. Ing. Filip Jurčíček Ph.D.

Study programme: Informatics

Specialization: Theoretical Computer Science

Prague 2015

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.


In ........ date ............                    signature of the author

Název práce: Development of a cloud platform for automatic speech recognition

Autor: Ondřej Klejch

Katedra: Ústav formální a aplikované lingvistiky

Vedoucí diplomové práce: Mgr. Ing. Filip Jurčíček Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Tato diplomová práce představuje cloudovou platformu pro automatické rozpoznávání řeči, CloudASR, která je postavena na systému pro rozpoznávání řeči Kaldi. Platforma podporuje dávkový a online způsob rozpoznávání řeči a také obsahuje anotační prostředí pro přidávání přepisů k odeslaným nahrávkám. Mezi klíčové vlastnosti této platformy patří škálovatelnost, přizpůsobitelnost a jednoduchý proces nasazení. Provedená měření dokázala, že latence platformy je porovnatelná s latencí Google Speech API a přesnost přepisů na omezených doménách může být dokonce lepší. Dále bylo ukázáno, že je platforma schopná zpracovat více než 1000 paralelních dotazů, pokud má dostatek výpočetních zdrojů.

Klíčová slova: cloud, Kaldi, ASR

Abstract: This thesis presents a cloud platform for automatic speech recognition, CloudASR, built on top of Kaldi speech recognition toolkit. The platform supports both batch and online speech recognition mode and it has an annotation interface for transcription of the submitted recordings. The key features of the platform are scalability, customizability and easy deployment. Benchmarks of the platform show that the platform achieves comparable performance with Google Speech API in terms of latency and it can achieve better accuracy on limited domains. Furthermore, the benchmarks show that the platform is able to handle more than 1000 parallel requests given enough computational resources.

Keywords: cloud, Kaldi, ASR

# Contents

# Introduction

Speech is the most natural form of human communication. In order to be able to talk with a computer, it is crucial to have a good Automatic Speech Recognition (ASR) system. On one hand, there are several open-source ASR toolkits, however deployment of such toolkits requires substantial knowledge, which makes them difficult to use for common software developers. On the other hand, there are a few web services that provide ASR, yet these web services do not solve all problems - either they are paid, closed-source or they are not customizable. So **the first goal of the present thesis is to develop a cloud platform for ASR** that is easy to use both from user's and maintainer's point of view.

Although the quality of ASR systems is improving, these systems are still far from perfect. One of the reasons is that the quality of ASR systems depends heavily on the amount of the training data, and there is not enough publicly available transcribed speech data for all languages. By providing free ASR web service it is possible to collect vast amount of recordings that can be manually transcribed. Consequently, **the second goal of the present thesis is to create an annotation interface** so that recordings obtained by CloudASR platform can be annotated and given back to the community.

The following text development and deployment of CloudASR platform and its annotation interface are described. Chapter 1 introduces Automatic Speech Recognition theory and tools related to CloudASR. In Chapter 2, tools used for CloudASR development and deployment are presented. The implementation of CloudASR platform is described in Chapter 3. Chapter 4 contains results of conducted benchmarks. Finally, Chapter 5 concludes this thesis.

# 1. Theoretical Background

This chapter describes theory needed for the CloudASR platform. It starts with **Automatic Speech Recognition (ASR)** section, which introduces concepts such as acoustic models, language models, speech decoding or evaluation of ASR systems. After that, **Voice Activity Detection** is described. Then, sections **Open-Source ASR Tools** and **Public ASR services** present technologies related to CloudASR. Finally, this chapter ends with **Obtaining Manual Transcriptions** section, which shows how manual transcriptions can be obtained with crowd–sourcing.

## 1.1 Automatic Speech Recognition

The task of the automatic speech recognition system is to "pick the most likely word sequence $\widehat{W}$ given the observed acoustic evidence A" [17]. Therefore, speech recognition can be described as a following formula:

$$\widehat{W} = \arg\max_W P(W|A) \tag{1.1}$$

By using Bayes' formula of probability theory, right-hand side of the Equation 1.1 can be rewritten in a following way:

$$\widehat{W} = \arg\max_W \frac{P(A|W)P(W)}{P(A)} \tag{1.2}$$

Since the numerator P(A) is constant regarding the maximization, it can be omitted to get the final equation:

$$\widehat{W} = \arg\max_W P(A|W)P(W) \tag{1.3}$$

Then, the probability $P(A|W)$ is called acoustic model and the probability $P(W)$ is called language model. In the sections **Acoustic Models** and **Language Models** computation of these probabilities will be described in more detail. After that, **Speech Decoding** section will describe how the Formula 1.3 is used to find the desired word sequence.

### 1.1.1 Acoustic Models

The role of acoustic models $P(A|W)$ is to capture all acoustic conditions, such as pronunciation, background noise, reverberation or transmission channel conditions, for all possible pairings of W and A. Traditionally, Hidden Markov Models (HMM) are used for acoustic modelling, but there are also other approaches based on artificial neural networks [23] or on dynamic time warping [32].

The acoustic model for a word sequence is a concatenation of HMM models for the individual words, which belong to the recognized vocabulary. These smaller models are built from the smaller HMMs for the basic building blocks of the acoustic model systems - phonemes.

Training of the acoustic models, as described in [17], consists of several steps. First, pronunciation, made up of the phonetic alphabet $\phi$, of each word in the

vocabulary is added to the phonetic dictionary. Note, that some words may have several pronunciations, then every valid pronunciation has to be added to the phonetic dictionary. Second, an elementary HMM is created for each symbol of the phonetic alphabet $\phi$. Third, for each word in the vocabulary a HMM is created as concatenation of elementary HMMs according to the word pronunciation. After that, these HMMs are concatenated with elementary HMMs corresponding to silence and/or end of words symbols to make a composite model for a transcription of some words sequence. Finally, these HMMs are trained with Baum–Welch [37] algorithm, which is a variation of EM algorithm [3], on recordings with theirs transcriptions.

Another way to train HMMs is to used Viterbi training algorithm [7], which approximates EM algorithm by choosing single best alignment and maximizing the posterior probability for the chosen alignment. Furthermore, latest work shows that Viterbi training achieves the same performance as Baum-Welch algorithm with much less computational resources [33].

### 1.1.2 Language Models

The task of the language model $P(W)$ in speech recognition is to determine how likely are the sequences of words $w_1, \ldots, w_m$ that sound alike by assigning a probability to each sequence. Using the Bayes' rule, $P(W)$ can be seen as:

$$P(W) = P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i | w_1, \ldots, w_{i-1}) \tag{1.4}$$

Since the probability $P(w_i | w_1, \ldots, w_{i-1})$ has just too many arguments and the probability does not necessarily depends on the entire history, the history is put into equivalence classes $\phi(w_1, \ldots, w_{i-1})$. This results into the following formula:

$$P(w_1, \ldots, w_m) \approx \prod_{i=1}^{m} P(w_i | \phi(w_1, \ldots, w_{i_1})) \tag{1.5}$$

Traditionally, n-gram language models, which use the following history equivalence classes $\phi(w_1, \ldots, w_{i-1}) = w_{i-(n-1)}, \ldots, w_{i-1}$, are used in speech recognition tasks. Thus, the n-gram language model becomes:

$$P(w_1, \ldots, w_m) \approx \prod_{i=1}^{m} P(w_i | w_{i-(n-1)}, \ldots, w_{i-1}) \tag{1.6}$$

Where the probabilities $P(w_i | w_{i-(n-1)}, \ldots, w_{w-1})$ are estimated from the relative frequencies of n-grams in the training data with the following formula:

$$P(w_i | w_{i-(n-1)}, \ldots, w_{i-1}) = \frac{c(w_{i-(n-1)}, \ldots, w_i)}{c(w_{i-(n-1)}, \ldots, w_{i-1})} \tag{1.7}$$

Since the numerator of the Equation 1.7 can be zero due to data sparsity problem, several smoothing techniques such as Jelinek–Mercer [16], Good–Turing [8] or Kneser–Ney [19] are often used to estimate the higher n-gram relative frequencies from the lower n-gram frequencies.

Recently, artificial neural networks have been also successfully used to tackle the data sparsity problem [1]. For instance, the recurrent neural network based

language models yielded state-of-the-art results in terms of WER in speech recognition [27].

### 1.1.3 Speech Decoding

Speech decoding is used to find the most likely word sequence $\widehat{W}$ using the Equation 1.3. Because the space of all word sequences is astronomically large, the search cannot be done by brute force.

One of the algorithm that can be used for speech decoding is Viterbi algorithm [4]. It is a dynamic programming algorithm for finding the most likely sequence of hidden states in HMMs. Even though it is not guaranteed that this algorithm will find the most likely word sequence $\widehat{W}$, it achieves very good results.

Viterbi algorithm has a problem with large vocabularies, because the resulting HMM has just too many states. As a result, states have to be pruned. There are several ways how to do that [15]. First, only top n states with the highest probabilities are kept. Second, only states with probability higher than the threshold from the maximal state probability of this frame are kept. Or a combination of these two methods can be used.

### 1.1.4 Evaluation

**Word error rate (WER)** is the most common metric used to evaluate the performance of ASR systems. It is computed as a minimum edit distance on words between one-best ASR output and a reference transcription divided by the number of words in the reference transcription.

$$WER = \frac{S + D + I}{N} \tag{1.8}$$

where:

- S is the number of substitutions,

- D is the number of deletions,

- I is the number of insertions,

- N is the number of words in reference

Metrics that are used to evaluate speed of ASR systems are **Real Time Factor (RTF)** and **Latency**. RTF is computed as a time needed to process the recording R by the ASR system divided by the length of the recording R. Latency is a delay between the end of the recording and the end of the recognition.

$$RTF = \frac{time(decode(R))}{length(R)} \tag{1.9}$$

## 1.2 Voice Activity Detection

Voice Activity Detection (VAD) is a technique used to detect presence or absence of human speech in the recording. There are several ways how to implement VAD, for example, support vector machines [26], gaussian mixture models [28] or deep neural networks (DNN) [34] can be used. The last one, using deep neural networks, will be described here.

The VAD using deep neural networks starts by extracting features every 10ms from a 25 ms analysis window. These features are then normalized and concatenated to a feature vector, which is used as an input to the DNN. The DNN then returns the posterior probability of speech being present in the analysis windows. This posterior probability is compared with a threshold, that is chosen so that false alarm and miss rates are equal on the test set, in order to yield decision whether the analysis window contains speech/non-speech.

## 1.3 Open-Source ASR Tools

In the following section some of the open-source ASR tools will be described. Namely, **HTK**, **Julius**, **Kaldi** and **RWTH**.

**HTK** [39] is the first toolkit, that will be described. HTK is a toolkit for building and manipulating hidden Markov models. It consists of a set of library modules and tools that provide sophisticated facilities for speech analysis, HMM training, testing and results analysis. Furthermore, it supports HMMs using both continuous density mixture Gaussians and discrete distributions and can be used to build complex HMM systems.

The second toolkit is **Julius** [21], high-performance large vocabulary speech recognition decoder, that can perform almost real-time decoding with 60k words in the vocabulary. It supports statistical n-gram language model and rule-based grammars, as well as Hidden Markov Model (HMM) as an acoustic model Julius can be also used with models trained for HTK toolkit.

Next described toolkit is **Kaldi** [31] – a free, open-source toolkit for speech recognition written in C++. Its speech recognition system is based on finite-state transducers and it supports modelling of arbitrary phonetic-context sizes, acoustic modelling with subspace Gaussian mixture models (SGMM) as well as standard Gaussian mixture models and deep neural networks, together with all commonly used linear and affine transforms.

Furthermore, there is also a Python wrapper for Kaldi called PyKaldi [30], which supports the online speech recognition. CloudASR uses PyKaldi as a default speech recognition system.

The last toolkit that will be described is **RWTH** [35], which is a publicly available speech recognition toolkit developed at Aachen University. It includes state of the art speech recognition technology for acoustic model training and decoding. Besides, its notable components are speaker adaptation, speaker adaptive training, unsupervised training, a finite state automata library, and an efficient tree search decoder.

## 1.4 Public ASR services

In addition to these open sources ASR toolkits there are also several web services that provide an API for speech recognition. Some of these services will be described in the following section.

**Google Speech API** supports speech recognition for 39 languages and their dialects. Its batch API, illustrated in Figure 1.1, is very simple and can be used for transcription of the wave or flac files. Additionally, Google Speech API supports the online speech recognition mode through JavaScript class SpeechRecognition in Google Chrome web browser.[1]

```
curl -X POST --data-binary @recording.wav \
    --header 'Content-Type: audio/x-wav; rate=16000;' \
    'https://www.google.com/speech-api/v2/recognize?lang=en-gb'
```

**Figure 1.1:** *An example of Google Speech API batch speech recognition mode request for a transcription of a recording in British English.*

**Nuance Dragon NaturallySpeaking**[2] is the second provider of the API for speech recognition. It provides software development kits for Windows and mobile applications. It also has a version that can be deployed on a server and used as a API for other applications.

The last API provider that will be mentioned here is **wit.ai**[3]. It supports 11 languages via an API similar to Google Speech API, see Figure 1.2 for an example, and in addition to speech recognition it also supports intent classification of the submitted recordings, see Figure 1.3 for an exemplar response from the wit.ai.

```
curl -X POST --data-binary @recording.wav \
    --header 'Content-Type: audio/x-wav; rate=16000;' \
    'https://www.google.com/speech-api/v2/recognize?lang=en-gb'
```

**Figure 1.2:** *An example of wit.ai API request for a transcription of a recording.*

## 1.5 Obtaining Manual Transcriptions

A large amount of transcribed recordings is needed in order to train a good ASR system. But a manual transcription of the recordings by professional transcribers is expensive and time demanding, typically, professional transcribers need 6 hours to transcribe 1 hour of speech data [38]. Furthermore, it is difficult to find enough professional transcribers to transcribe the required amount of speech data in short time.

Recently, it was shown that crowd-sourcing can be used for cheap, fast and good enough manual transcription of speech data [29]. With crowd-sourcing,

---

[1] https://www.google.com/intl/en/chrome/demos/speech.html
[2] http://www.nuance.com/for-developers/dragon/index.htm
[3] https://wit.ai/

```
{
    "msg_id" : "e83406f9-238c-40c8-9a99-6913b33b4301",
    "_text" : "I'm looking for a bar",
    "outcomes" : [ {
      "_text" : "I'm looking for a bar",
      "intent" : "restaurantSearch",
      "entities" : {
        "search_query" : [ {
          "suggested" : true,
          "value" : "bar",
          "type" : "value"
        } ]
      },
      "confidence" : 0.912
  } ]
}
```

**Figure 1.3:** *An exemplar response from wit.ai API with a recording of a sentence: "I'm looking for a bar."*

speech data is split into small chunks that are then transcribed by several non-professional transcribers. Their transcriptions are then used to select the best transcription for the recording, for example with ROVER algorithm [24], and used for training of new ASR systems. Also, transcriptions from non-professional transcribers are only 6% worse than professional transcriptions and they cost only $\frac{1}{30}$ of the cost of professional transcription. Finally, services like **Amazon Mechanical Turk**[4] or **CrowdFlower**[5] already support the speech transcription tasks.

---

[4]https://www.mturk.com
[5]http://www.crowdflower.com/

# 2. Used Technologies

In this chapter technologies that were used during development will be described. Also the motivation for the usage of these technologies will be explained.

## 2.1 Platform

In the following section technologies that were used to build a cloud platform will be described. These technologies have made it possible to build a scalable solution with an easy deployment.

Traditionally, a deployment of a such a complex system as CloudASR consists of several steps during which necessary dependencies are installed, the application environment is set up and finally the application is started. But this approach makes the maintenance of these systems difficult, because the deployment is time consuming, error-prone and it is not replicable. The ultimate goal for the CloudASR deployment was the exact opposite: fast and replicable deployment.

The most important tool used during the development was **Docker** [25] – a portable, lightweight application runtime and packaging tool. It allows to specify dependencies and environmental variables for a process and it allows to build an image from this specification called Dockerfile (see Figure 2.1 for example). Once this image is built it can be used on any machine with Docker installed, which makes the deployment fast and replicable, because it is not necessary to install all dependencies on every machine. Additionally, the usage of Docker images removes bugs caused by different versions of libraries used in development and production environment because developers use the same images in both environments.

```
FROM ubuntu
MAINTAINER Ondrej Klejch

RUN sudo apt-get update && sudo apt-get install python
ADD . /opt/app
WORKDIR /opt/app

CMD python run.py
```

**Figure 2.1:** *An example of Dockerfile that creates an image from the base Ubuntu image, installs python, copies all files in the Dockerfile folder and sets command python run.py to be run, when the docker image is started.*

When running an application in the cloud it is neccessary to monitor all servers and handle failovers. But with an increasing number of servers, maintenance costs grow rapidly. Therefore, it is not possible to manage the application manually. The tool that allows CloudASR to run on many servers is **Mesos** [12]. It lets users program against a set of machines in the same way as if it was a single machine, which means that it is possible to run and scale an application on a set of servers in a similar way as on a single machine. Mesos takes care of the scheduling and high availability of the platform. Thus, whenever some part of the

**Figure 2.2:** *A screenshot of a Marathon web interface with a running CloudASR platform.*

CloudASR crashes, Mesos will try to restart it. Finally, Mesos supports Docker so the images that are used in development can be also used on a Mesos cluster.

**Marathon**[1] is a framework built on top of Mesos whose main responsibility is to launch long running applications. It is an entrypoint for running and scaling the applications running on a Mesos cluster. It has a web user interface (see Figure 2.2) and a REST API, through which applications can be started, scaled or stopped easily.

Since the traffic of CloudASR platform can be very large, it is not possible to process all HTTP requests by one application server. Therefore, CloudASR platform uses **HAProxy**[2] load-balancer to distribute workload between application servers, but any other load-balancers can also be used with appropriate setup.

## 2.2 Continuous Integration & Delivery

Several practises were obeyed during the development, namely Continuous Integration and Continuous Delivery. For this a platform which consisted of **Jenkins-CI**[3] and **Docker Registry**[4] was deployed.

The most important tool for Continuous Integration & Delivery of CloudASR is Jenkins-CI. Its task is to watch CloudASR git repository and whenever a new code is pushed into this repository it schedules a new build of the platform. During this build, the most recent code is pulled from the repository and then the new docker images are built. After that tests are run to check that the new code did not break anything. Finally, successfully built images are tagged with current build number and pushed to the Docker Registry.

Docker Registry is a repository of Docker images. Even though, there are sev-

---

[1] https://mesosphere.github.io/marathon/
[2] http://www.haproxy.org/
[3] https://jenkins-ci.org/
[4] https://github.com/docker/docker-registry

eral Docker Registry providers[5], which are free for open-source projects, CloudASR uses its own free Docker Registry in order to be able to use also proprietary software that cannot be shared with public.

## 2.3   Backend

The main programming language used for backend development is **Python**[6]. Web is built on top of **Flask**[7] microframework and it uses **Gunicorn**[8] for production deployment. **MySQL**[9] is used as a database, but any other SQL database can be used instead, because the database is accessed through **SQLAlchemy**[10].

The CloudASR architecture consists of several nodes which need to communicate between each other. CloudASR uses **ZeroMQ**[11] for this communication because of its simple design, high performance and support for every modern language. With ZeroMQ, it is possible to create many messaging patterns, but CloudASR uses only two: request-reply and push-pull. In the request-reply messaging pattern a sender sends a message and then waits for a reply, after that another sender can send next message and wait for a reply. On the other hand, in the push-pull messaging pattern senders just send messages and do not wait for replies.

In order to be able to send complex messages via ZeroMQ sockets, messages have to be serialized. CloudASR uses **Google Protocol Buffers**[12], because they have support in many languages, allow specification of various message types (See Figure 2.3 for example) and serialize messages in very compact way (See Table 2.1 for a comparison of different serializations).

| raw file size | 56146 | |
|---|---|---|
| bytes_protobuf | 56118 | 0.999x |
| base64 | 74872 | 1.333x |
| json_array | 158590 | 2.824x |

**Table 2.1:** *The table shows comparison of different serialization used to serialize a wave file into a message for the CloudASR online mode. As can be seen from the results Google Protocol Buffers achieved the best result.*

CloudASR uses **Pykaldi** [30] as a Python wrapper for the **Kaldi speech recognition toolkit** [31]. Because CloudASR should be able to process very long recordings, possibly infinite, with limited computational resources, it is necessary to split the recordings into smaller chunks. For that purpose CloudASR uses voice activity detector implemented in **Theano** [2] to detect silences in a speech.

---

[5]https://hub.docker.com/, https://quay.io/
[6]https://www.python.org/
[7]http://flask.pocoo.org/
[8]http://gunicorn.org/
[9]https://www.mysql.com/
[10]http://www.sqlalchemy.org/
[11]http://zeromq.org/
[12]https://developers.google.com/protocol-buffers/

```
message HeartbeatMessage {
  required string address = 1;
  required string model = 2;
  required Status status = 3;

  enum Status {
    STARTED = 0;
    WAITING = 1;
    WORKING = 2;
    FINISHED = 3;
  };
}
```

**Figure 2.3:** *An example of Google Protocol Buffer message specification with three fields. Fields address and model are just strings and the status is an enum with four possible values.*

## 2.4 Frontend

The frontend uses several well-known open-source libraries, namely, **Twitter Bootstrap**[13] for CSS styling of the web, **jQuery**[14] and **Angular.js**[15] for interactive elements on the web.
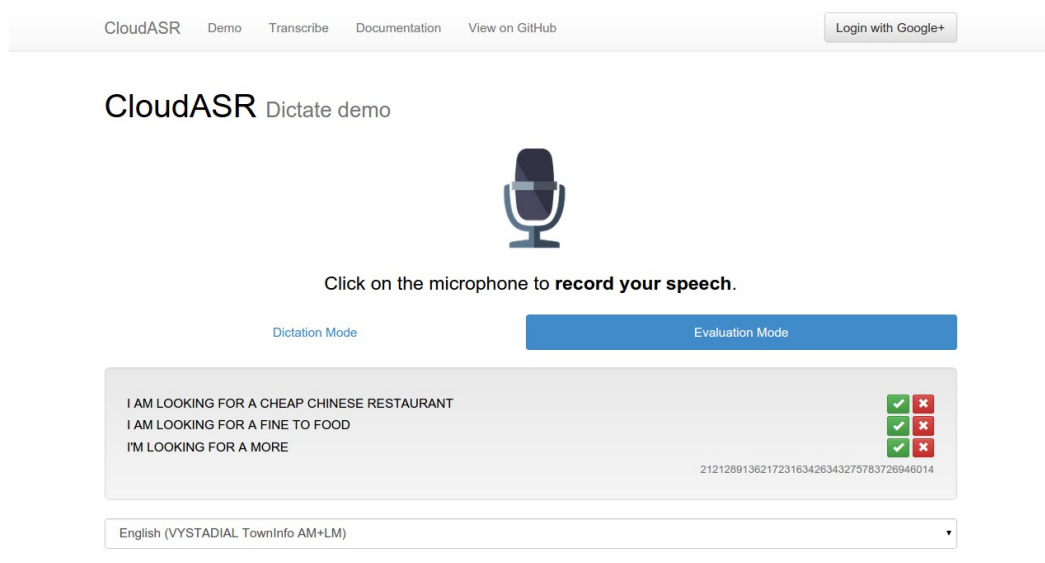


**Figure 2.4:** *Screen of the Web Demo*

Modern web browsers support **WebAudio API**[16], which is a high-level JavaScript API for processing and synthesizing audio in web applications. One of the things that can be done with this API is audio recording. Thus, it is possible to create a

---

[13]http://getbootstrap.com/2.3.2/
[14]https://jquery.com/
[15]https://angularjs.org/
[16]http://webaudio.github.io/web-audio-api/

web demo for the CloudASR online mode. The demo is based on **Recorder.js**[17] library, which can record output of WebAudio API and return it as a PCM chunks.

The next step is to send these chunks to the API. Because the demo demonstrates the online speech recognition mode, it is not possible to wait for the whole recording to be recorded and then send it to the API via HTTP POST request. Thus, CloudASR uses **Socket.IO**[18] to send stream of chunks to the API and to receive stream of results from the API.

---

[17]`https://github.com/mattdiamond/Recorderjs`
[18]`http://socket.io/`

# 3. Implementation

This chapter describes the implementation of the CloudASR platform. The platform provides API for both batch and online speech recognition mode and it has an annotation interface for adding transcriptions to submitted recordings. The implementation was also affected by the following requirements:

- **Scalability** - because the speech recognition is a demanding process in terms of computational resources, it is not possible to handle many parallel requests on one machine. Therefore the CloudASR architecture had to be designed to be able to scale across many machines.

- **Easy deployment** - complex systems as CloudASR is have many dependencies and a difficult deployment process, which makes their maintenance hard. CloudASR should have as few dependencies as possible and one command deployment.

- **Customizability** - there are already several web services that provide an API for speech recognition, but they are not easily customizable. Thus, the second requirement was to be able to host any Kaldi model on the CloudASR platform. Moreover, the CloudASR platform should be able to run any ASR system, if the users implement a wrapper for that system.
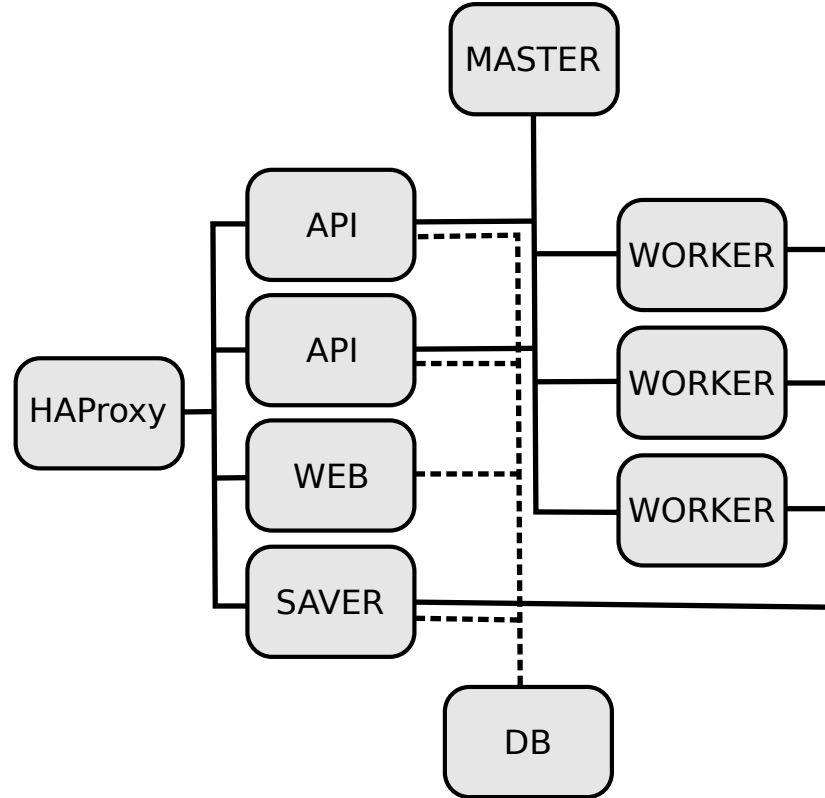
## 3.1 Architecture

In order to meet the aforementioned scalability requirement the platform had to be designed from the very beginning to be able to run on many machines. As a result, the architecture consists of several nodes that communicate with each other by sending messages over ZeroMQ sockets (each node acts as an actor as described in [11]).

The architecture was also affected by the fact that it is not possible to start an ASR system when the user sends a request, because the ASR systems need some time to load decoding graphs in the memory, which would add some unnecessary latency. To solve this problem the platform uses Master-Worker architecture, which also makes it possible to handle requests for various languages at the same time.

The CloudASR architecture as described in Figure 3.1 consists of several types of nodes that can run on different machines. These nodes are **Master**, **Worker**, **API**, **Web**, and **Recordings Saver**. In the following section each node will be described in detail.

### 3.1.1 Master

The main task of Master is to keep track about running workers and to schedule tasks to them. In order to be able to handle requests for various languages Master monitors state of each worker and it has a queue of waiting workers for each language, so that when API asks for a worker Master can return an address of an available worker.

**Figure 3.1:** *An overview of the CloudASR architecture. The most important node is* **Master** *to which* **Workers** *sends heartbeats with their state. The client requests are handled by* **API** *which communicates with Master and Workers. The processed recordings are sent to* **Saver** *which saves them and serves them via HTTP. An annotation interface and an online demo are hosted by* **Web**. *Finally, there are also two external nodes:* **HAProxy** *which load-balances requests between particular application instances and* **DB** *which stores information about processed recordings.*

The workers can be in four different states: **started**, **waiting**, **working** and **not responding** and they send four different heartbeats, small messages with an information about their state, to Master: **started**, **waiting**, **working** and **finished**.

The life cycle of the worker as described in Figure 3.2 starts in the **started** state, after that it moves to the **waiting** state by sending the **waiting** heartbeat. The worker remains in the waiting state until **Master assigns a tasks** to it, then it moves to the **working** state where it remains as long as it is working. In the working state worker sends working heartbeats periodically, to inform Master that it is working and it did not fail. At the end of the task the Worker sends **finished** heartbeat and Master changes the state of the Worker to the **waiting** state.

Additionally, when a worker crashes during the processing of the task and it gets restarted, it sends started heartbeat again, which informs Master, that the worker was restarted and it adds it to the queue again. When a worker does not send any heartbeat for 10 seconds, the master sets the worker state to **not responding**. But as soon as the worker sends any heartbeat, the master will set the worker to the appropriate state.

Unfortunately, Master is a single point of failure of the CloudASR platform.
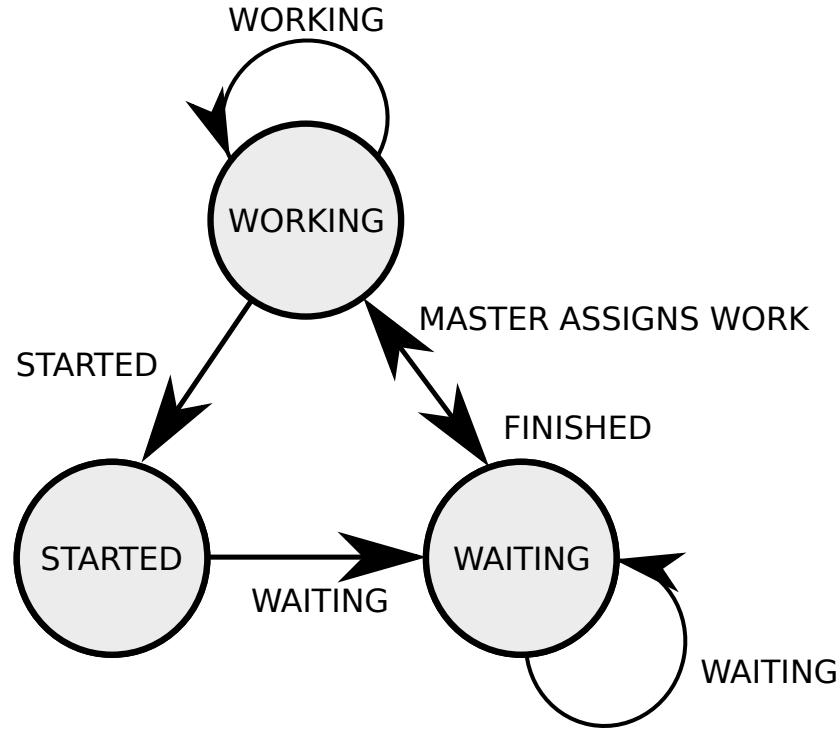
**Figure 3.2:** *The life cycle of the worker starts in in the **started** state, after that it moves to the **waiting** state by sending the **waiting** heartbeat. The worker remains in the waiting state until **Master assigns a tasks** to it, then it moves to the **working** state where it remains as long as it is working. In the working state worker sends working heartbeats periodically, to inform Master that it is working and it did not fail. At the end of the task the Worker sends **finished** heartbeat and Master changes the state of the Worker to the **waiting** state.*

When Master stops working no speech recognition requests can be processed, because the API containers will not know to which worker they can forward the request. But as soon as Master starts working the platform should be available again.

### 3.1.2 Worker

Worker acts as a wrapper for an ASR system. By default Pykaldi is used but any other ASR system can be used if the user implements a python wrapper for that system. Because CloudASR should be able to process very long recordings in the online speech recognition mode, it is necessary to split the recordings into smaller chunks that can be processed with limited computational resources. For that purpose VAD component from Alex Statistical Dialogue Systems Framework [18] is used to detect silence in a speech, because the speech can be split at that point without any large negative effect on the accuracy of the transcriptions.

### 3.1.3 API

The main task of API is to forward requests from the clients to the workers. When API receives a request from a client, it sends a message to Master with a request for a worker address for the given language. If there is any available

worker, Master returns its address, otherwise Master returns an error which is forwarded back to the client. Then API sends the submitted recording to the worker, which processes it and sends back either interim results for the online speech recognition mode or final results for the batch speech recognition mode. Finally, API sends a response with the results to the client.

The API is built on top of Flask framework with enabled asynchronous processing which allows single API container to process many parallel requests, because there are no blocking operations in the API container - it just receives requests from clients and forwards them via ZeroMQ to the workers.

### 3.1.4 Web

CloudASR platform has a web interface with an online demo and an annotation interface. The online demo (See Figure 3.3) allows users to try out CloudASR directly in their web browsers. It has two modes, namely, dictation mode, which only shows the best transcription of the recording, and evaluation mode, which also allows users to confirm that the transcription of the recording is correct.
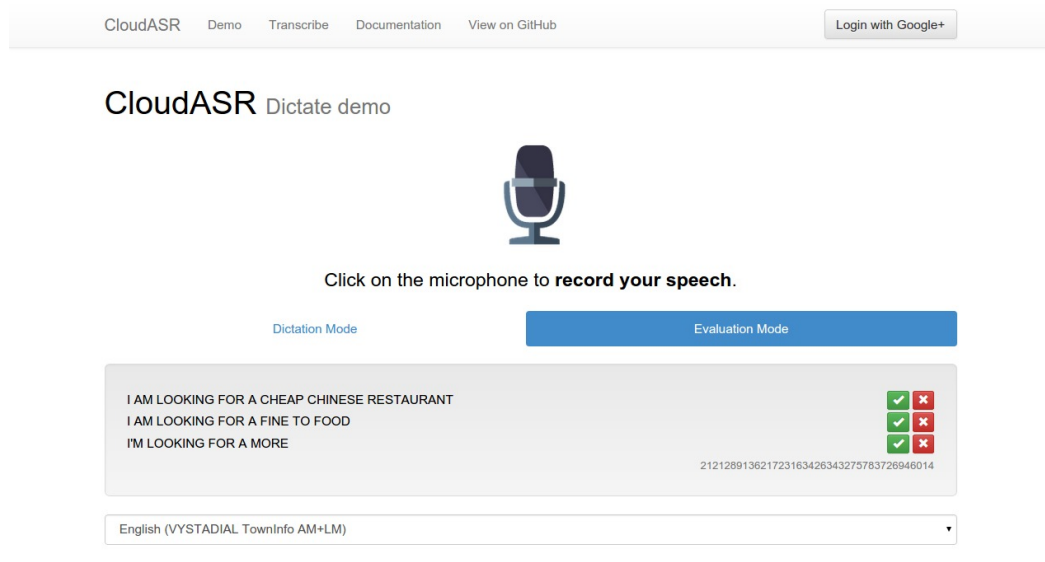


**Figure 3.3:** *Screen of the Web Demo*

Even though the annotation interface allows anonymous users to add transcriptions to recordings, the users are encouraged to log in via their Google Account, because then it is possible to track their transcriptions and make useful insights about the quality of their transcriptions.

The annotation interface distinguishes between two types of user roles: users and administrators. Normal users are only allowed to add transcriptions (See Figure 3.4) to the recordings selected by CloudASR, but administrators can view every recording with its transcriptions.

In order to get the most of the normal user transcriptions, a recording, which should be transcribed by the user, is selected randomly from all recordings for the given language with confidence lower than 0.8, because it is not necessary to transcribe recordings with correct transcriptions. This allows to collect transcriptions for all recordings uniformly and then it is up to the administrator to decide which transcription is the best.
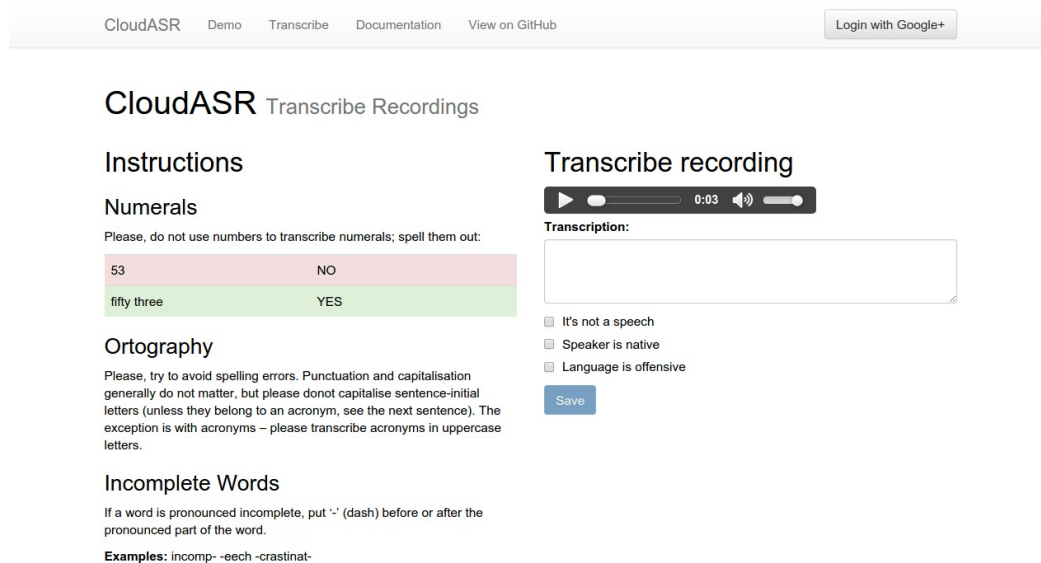
**Figure 3.4:** *Screen of the Annotation Interface*

### 3.1.5 Recordings saver

The main task of the Recordings saver is to save and serve recordings processed by workers. When the worker finishes recognition it sends the recording with its n-best hypotheses to the Recording saver via ZeroMQ socket. The saver saves the wave file to the file system and it saves the n-best hypotheses to the database so that they can be used in the future.

## 3.2 Scalability

As mentioned before, one of the key requirements for the CloudASR platform was scalability. To successfully fulfill this requirement architecture was split into smaller nodes, which communicate which each other by sending messages over ZeroMQ sockets. This enables the platform to run on several machines. Additionally, the heartbeating makes it possible to scale workers dynamically without need to stop the platform. Finally, load-balancing allows to run several API nodes and spread the load between. Yet, this solution is limited by the network capacity, but it is possible to deploy the CloudASR to a different data center and load balance between data centers. This makes the CloudASR platform almost infinitely scalable.

## 3.3 Deployment

The CloudASR platform supports two types of deployment: single host and multi host. Single host deployment allows users to run CloudASR directly on their machines with just one dependency installed - Docker. Whereas multi host deployment allows users to run CloudASR on a set of machines with Mesos installed.

Users can specify which workers they want to run in a configuration file, see Figure 3.5 for an example. In this file they can specify names of Docker images for the workers, number of instances of these workers and a model name with

18

which the worker will be available for speech recognition. Also, users have to specify runtime variables such as IP address of a slave where the Master should run, MySQL connection string and a domain name that HAProxy will use to route requests to the running platform. Finally, users have to specify credentials for Marathon if they want to run the platform on a Mesos cluster. After that they can run CloudASR locally with `make run_locally` command or on a Mesos Cluster with `make run_on_mesos` command.

```
{
    "domain": "cloudasr.klejch.eu",
    "marathon_url": "marathon_url",
    "marathon_login": "marathon_login",
    "marathon_password": "marathon_password",
    "master_ip": "master_ip",
    "connection_string": "mysql connection string",
    "workers": [
        {
            "image": "ufaldsg/cloud-asr-worker-cs-alex",
            "model": "cs-alex",
            "instances": 5
        }
        {
            "image": "ufaldsg/cloud-asr-worker-en-towninfo",
            "model": "en-towninfo",
            "instances": 5
        }
    ]
}
```

**Figure 3.5:** *An example of the CloudASR configuration `deployment/cloudasr.json` that specifies how to run 5 cs-alex workers and 5 en-towninfo workers using respective Docker image. With this configuration file the CloudASR platform can be run locally with command `make run_locally` or on a Mesos cluster with command `make run_on_mesos`.*

In order to ensure quality and stability of the CloudASR platform, two practises were used during the development: **Continuous Integration** [6] and **Continuous Delivery** [13]. The goal of these practises is to build, test and deploy the CloudASR platform as often as possible, to get a feedback from the real usage. To achieve that the CloudASR platform uses Jenkins-CI server, that watches the CloudASR git repository and on every push to the repository it builds Docker images, tests the code and push the built Docker images to the Docker Registry. After that it is possible to deploy the CloudASR platform with a specific version and it is also possible to switch back to the older versions when anything goes wrong. To minimize failures CloudASR is deployed first to the development environment, where the users can test it and then it can be deployed to the production environment.

To ensure a stability of CloudASR all crucial parts of the platform are covered with tests, namely unit tests, integration tests and end-to-end tests.

Each node was implemented with two main design patterns in mind, namely Dependency Injection [5] and Factory Method [9]. Usage of these patterns together with message oriented architecture made it possible to unit test the whole platform easily, because it enabled to pass test doubles into the nodes and then send fake messages needed to test a correct behaviour of the node. A typical unit test structure of the CloudASR platform node looks like a test in Figure 3.6.

In addition to unit tests there are also integration tests, that test the factory methods that create production ready nodes, and end-to-end tests, that test that both batch and online recognition mode requests are handled correctly. This test suite ensures that developers do not break anything and it also gives them confidence to change the code without fear.

```
def test_worker_sends_heartbeat_after_finishing_task(self):
    messages = [
        {"frontend": self.make_frontend_request("message 1")}
    ]

    self.run_worker(messages)
    self.assertThatHeartbeatsWereSent(["STARTED", "FINISHED"])
```

**Figure 3.6:** *An example of unit test that tests communication between nodes.*

## 3.4   Customizability

The second requirement for CloudASR is customizability in terms of acoustic and language models. The platform supports a creation of new workers with various acoustic and language models and it also supports a creation of new workers with arbitrary ASR systems. An overview on the customization process is described in the following section.

### 3.4.1   Worker with New Kaldi Models

In order to create a new worker with new Kaldi models a worker Docker image has to be made. The image is created in several steps. First, users have to create a script `download_models.sh` that will download all necessary files from their server, see Figure 3.7 for example. Second, they have to create a configuration file `config.py` with an appropriate configuration for the downloaded models, see Figure 3.8. Finally, they have to copy a Dockerfile (see Figure 3.9for example) for the worker and build the docker image with the appropriate command. After that users can use the new worker in their application in the similar way as they use other models.

### 3.4.2   Worker with Arbitrary ASR System

Even though CloudASR supports only Kaldi out of the box, other ASR systems can be used too. Again, the only thing that the users have to do is to create a worker docker image with their ASR system. The only step that differs from the

```bash
#!/bin/bash

DOMAIN=vystadial.ms.mff.cuni.cz
PATH=/download/alex/applications/PublicTransportInfoCS/hclg/models/
URL=https://$DOMAIN/$PATH

mkdir /opt/models
wget -O /opt/models/mfcc.conf $URL/mfcc.conf
wget -O /opt/models/tri2b_bmmi.mdl $URL/url/tri2b_bmmi.mdl
wget -O /opt/models/tri2b_bmmi.mat $URL/tri2b_bmmi.mat
wget -O /opt/models/HCLG_tri2b_bmmi.fst $URL/HCLG_tri2b_bmmi.fst
wget -O /opt/models/words.txt $URL/words.txt
```

**Figure 3.7:** *An example of download_models.sh script.*

```python
models_dir = '/opt/models'
wst_path = '%s/words.txt' % models_dir
kaldi_config = [
    '--config=%s/mfcc.conf' % models_dir,
    '--verbose=0', '--max-mem=10000000000',
    '--beam=12.0', '--lattice-beam=2.0',
    '--acoustic-scale=0.2', '--max-active=5000',
    '--left-context=3', '--right-context=3',
    '%s/tri2b_bmmi.mdl' % models_dir,
    '%s/HCLG_tri2b_bmmi.fst' % models_dir,
    '1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20',
    '%s/tri2b_bmmi.mat' % models_dir
]
```

**Figure 3.8:** *An example of config.py script.*

previous process is that the users have to implement and add to the Dockerfile a script `asr.py` with their own `create_asr` method that returns ASR class with these methods:

- `reset()` - this method is called after every request and it can be used to reset the underlying ASR system.

- `recognize_chunk(pcm)` - this method is used to process small chunks of recordings. The method should accept pcm chunks with frame rate 16000 and it should return an interim hypothesis in form of a tuple (`confidence`, `transcript`).

- `get_final_hypothesis` - this method is called at the end of every request, it should return a list of n-best hypotheses in form of a tuple (`confidence`, `transcript`).

The creation of such a script is illustrated in Figure 3.10 on the DummyASR class, which will aslo be used for benchmark purposes in the Chapter 4.

```
FROM ufaldsg/cloud-asr-worker
MAINTAINER Ondrej Klejch

WORKDIR /opt/app
ADD . /opt/app
RUN bash download_models.sh

ENV model cs-alex
```

**Figure 3.9:** *An example of worker Dockerfile.*

```
import time

def create_asr():
    return DummyASR()

class DummyASR:

    def recognize_chunk(self, chunk):
        time.sleep(float(len(chunk)) / 16000 / 2)
        return (1.0, 'Dummy interim result')

    def get_final_hypothesis(self):
        time.sleep(0.2)
        return [(1.0, 'Dummy final result')]

    def reset(self):
        pass
```

**Figure 3.10:** *An example of alternative ASR implementation.*

It is important to note that the new worker Docker images will be available only on the machine where they were built. If the users want to use these workers on multiple machines, they have to push them to their docker registry or they can update Jenkins scripts `build_workers.sh` and `push_workers.sh` and Jenkins will do that for them.

## 3.5   Example of API Usage

The CloudASR platform supports both batch and online speech recognition. In the batch mode users send wave files to the API using HTTP POST request and they receive a json with n-best transcriptions. Users can specify which worker they want to use in `lang` parameter. The batch mode has a similar interface to Google Speech API, which enables users to switch to CloudASR seamlessly. An example of batch recognition API usage is illustrated on a simple curl command in Figure 3.11 and a response from the API is shown in Figure 3.12.

In contrast to batch mode in online mode users send PCM chunks of a record-

```
curl -X POST --data-binary @recording.wav \
     --header 'Content-Type: audio/x-wav; rate=16000;' \
     'http://api.cloudasr.com/recognize?lang=en-towninfo'
```

**Figure 3.11:** *An example of batch speech recognition mode request for an en-towninfo worker using curl.*

```
{
  "result": [
    {
      "alternative": [
        {
          "confidence": 0.5549500584602356,
          "transcript": "I'M LOOKING FOR A BAR"
        },
        {
          "confidence": 0.14846260845661163,
          "transcript": "I AM LOOKING FOR A BAR"
        },
      ],
      "final": true
    }
  ],
  "result_index": 0
}
```

**Figure 3.12:** *An example of batch recognition mode response.*

ing while the recording is being recorder. Users can send these chunks as often as they want but it is advised to send a chunk four times per second to achieve a smooth experience. Chunks are sent to the server via Socket.IO technology encoded as JSON messages. Because JSON does not support encoding of a binary data, it is necessary to encode PCM chunks into a string. Base64 proved itself to be a sensible compromise between the message size increase and the implementation complexity. The CloudASR platform comes with a JavaScript library for online speech recognition mode. Figure 3.13 shows how this library can be used for speech recognition in Google Chrome web browser.

```
var speechRecognition = new SpeechRecognition();
speechRecognition.onStart = function() {
    console.log("Recognition started");
}

speechRecognition.onEnd = function() {
    console.log("Recognition ended");
}

speechRecognition.onError = function(error) {
    console.log("Error occured: " + error);
}

speechRecognition.onResult = function(result) {
    console.log(result);
}

var lang = "en-wiki";
$("#button_start").click(function() {
    speechRecognition.start(lang);
});

$("#button_stop").click(function() {
    speechRecognition.stop()
});
```

**Figure 3.13:** *JavaScript code that can be used for speech recognition in Google Chrome.*

# 4. Evaluation

In order to show that the CloudASR platform is ready for the production usage several benchmarks were made. First, real time factor (RTF) of the batch speech recognition mode was measured and compared with Google Speech API. Second, latency of the online speech recognition mode was measured. Finally, number of parallel requests for batch speech recognition mode was measured to show that CloudASR is scalable.

## 4.1  RTF of Batch Speech Recognition

In the first benchmark RTF of CloudASR batch mode was compared with RTF of Google Speech API using test set from the Czech Public Transportation Information Domain [20]. WER and RTF of both APIs were measured with following results: Google Speech API had 60% WER and RTF 0.3 whereas CloudASR had 22% WER and RTF 0.22. Request times of both APIs are displayed in Figure 4.1.

This benchmark shows that RTF of CloudASR batch mode is lower than RTF of Google Speech API. Moreover, the benchmark shows that the CloudASR platform can achieve better accuracy than Google Speech API on limited domains if the used decoding graphs are customized for those domains.
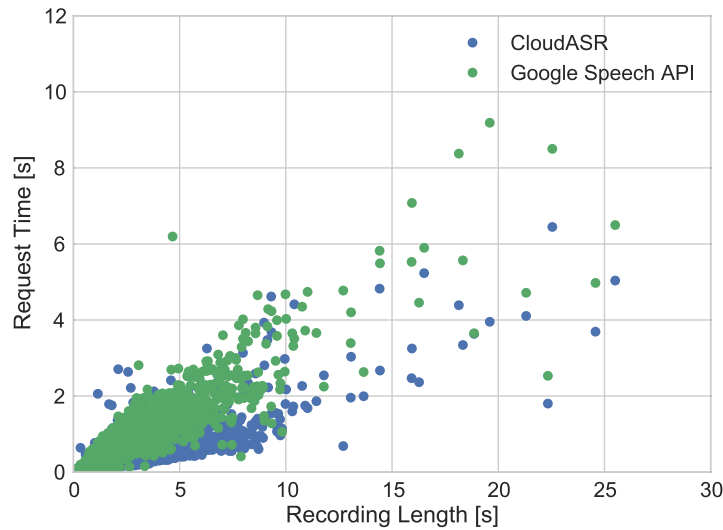


**Figure 4.1:** *Batch recognition benchmark.*

## 4.2  Latency of Online Speech Recognition

In the second benchmark latency of CloudASR online mode was measured. Low latency is crucial for successful usage of speech recognition in dialogue systems, but there are not so many web services that provide online speech recognition mode. Therefore, support for online speech recognition mode can be seen as a key feature of the CloudASR platform.

The reason why the online speech recognition mode is better suited for dialogue systems is that it is possible to get the results while the speech is being recorded. Thus, dialogue systems can react quickly. Results, plotted in Figure 4.2, show that the latency of the online speech recognition mode remains small even for long recordings which is in contrast to the increasing latency of the batch speech recognition mode.
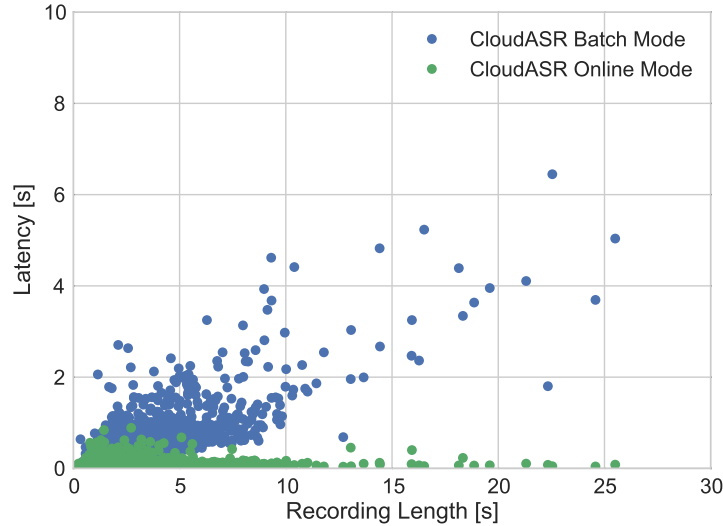


**Figure 4.2:** *A latency comparison for CloudASR online and batch mode.*

## 4.3 Parallel Requests Benchmark

Because the main bottleneck of the CloudASR platform is a number of running workers, workers with dummy ASR engine (described in Figure 3.10) were used to be able to test how many parallel requests can CloudASR handle. As a result, 1000 dummy workers could run on a Mesos cluster with 5 slaves (4CPU, 16GB RAM). Also, workload was spread across 5 frontend containers with a load-balancer.

```
seq 1 100 | xargs -P100 -I {} \
  curl -X POST --data-binary @resources/test.wav \
    --header "Content-Type: audio/x-wav; rate=16000;" \
    -s -w "%{time_total}\n" \
    http://api.cloudasr.com/recognize?lang=dummy
```

**Figure 4.3:** *A simple benchmark script that sends 100 parallel requests to the CloudASR API and prints out request time for each request.*

Then several benchmarks were run to show how RTF of the batch recognition mode changes with different number of parallel requests. The platform was tested with a different number of parallel requests (50, 250, 500, 750 and 1000) and with files with different lengths (5s, 10s, 20s, 30s, 40s, 50s and 60s). To be able to run
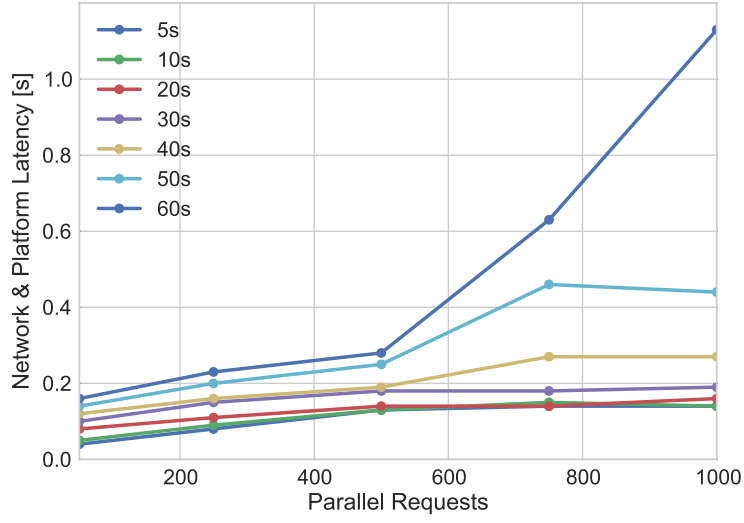
**Figure 4.4:** *The graph shows platform & network latency for recordings with various lengths given the number of parallel requests.*

so many parallel requests, a simple benchmark, shown in Figure 4.3, was run on 5 machines. Each machine sent one-fifth of the total number of parallel requests ten times and then the average latency was computed across all machines.

Results, summarized in Figure 4.4, show that CloudASR platform adds just a very little overhead compared to the raw dummy worker for small recordings. But the latency increases rapidly for large recordings with more than 500 parallel requests. This is probably caused by network capacity of the servers that run the benchmark and it should not affect the platform with the real usage. Moreover, this will not affect the online speech recognition mode because it sends only very small messages. Therefore, the platform should be able to handle much more parallel requests with appropriate number of workers.

# Conclusion

Goals of this thesis were to develop a cloud platform for ASR, CloudASR, and an annotation interface for annotating speech data. These goals were successfully accomplished. Furthermore, in addition to the original requirement to create batch speech recognition mode, online speech recognition mode was also implemented.

In the following sections all important results of this thesis are summarized and at the end ideas for future work are proposed.

## Cloud platform for ASR

The first goal of this thesis was to develop a cloud platform for ASR, CloudASR, that would provide API for batch speech recognition mode of the submitted wave files. This API is similar to Google Speech API, which enables users to switch to CloudASR seamlessly. In addition, CloudASR provides an API for online speech recognition. The CloudASR comes with a web demo, where the users can try out the online speech recognition in various languages. Furthermore, the platform is scalable, customizable and easily deployable.

In terms of scalability, the platform is able to run both on a single machine and a multi-machine setup and it allows to scale the number of running workers according to the users' needs. The benchmarks show that the platform is able to handle more than 1000 parallel requests.

The platform can handle requests for various languages at the same time. Users can create workers for new languages using Pykaldi or they can even create workers for an arbitrary ASR systems if they provide a Python wrapper for that system.

CloudASR is easily deployable, it uses Docker for creating and running application containers on a single machine and it uses a Mesos cluster to run CloudASR on multiple machines.

## Annotation interface

The second goal of this thesis was to create an annotation interface for annotating submitted recordings. Its responsibility is to collect and store submitted speech recordings together with their transcriptions. Then users can rate the automatic transcriptions of the recordings or they can provide their own transcriptions if they think that none of the automatic transcriptions is correct. The annotation interface allows the administrators to choose the best transcription from several manual transcriptions that were obtained for the recording. Additionally, external service, such as CrowdFlower, can be used to obtain manual transcriptions of the selected recordings.

## Future work

- Since manual transcription of recordings is expensive it would be good to make users transcribe only parts of the recordings in which ASR system

was not confident enough [36]. This idea could be used for both user transcription and CrowdFlower transcription.

- With manually transcribed recordings from CloudASR platform it is possible to continuously improve the accuracy of the underlying ASR system by adapting the language model to the type of language that the users of the CloudASR really use. Thus CloudASR could provide an option to automatically update language model when a certain amount of new transcribed recordings was collected.

- Because running CloudASR platform is expensive in terms of the costs for the server hosting, it would be good to optimize the usage of the individual workers. Spare workers should be shut down when there is no need for them and new workers should be started when the traffic arise. This can be achieved either by a feedback control based system [14] or by using machine learning techniques [10].

- As CloudASR platform provides an API for speech recognition, it could also be used for another speech related tasks like Language Identification, Speaker Identification or Voice Activity Detection.

# Bibliography

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin, *A neural probabilistic language model*, The Journal of Machine Learning Research **3** (2003), 1137–1155.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio, *Theano: A CPU and GPU math compiler in Python*, Proc. 9th Python in Science Conf, 2010, pp. 1–7.

[3] Arthur P Dempster, Nan M Laird, and Donald B Rubin, *Maximum likelihood from incomplete data via the em algorithm*, Journal of the royal statistical society. Series B (methodological) (1977), 1–38.

[4] G David Forney Jr, *The viterbi algorithm*, Proceedings of the IEEE **61** (1973), no. 3, 268–278.

[5] Martin Fowler, *Inversion of control containers and the dependency injection pattern*, 2004.

[6] Martin Fowler and Matthew Foemmel, *Continuous integration*, Thought-Works) http://www.thoughtworks.com/Continuous Integration. pdf (2006).

[7] Michael Franzini, K-F Lee, and Alex Waibel, *Connectionist viterbi training: A new hybrid method for continuous speech recognition*, Acoustics, Speech, and Signal Processing, 1990. ICASSP-90., 1990 International Conference on, IEEE, 1990, pp. 425–428.

[8] William Gale and Geoffrey Sampson, *Good-turing smoothing without tears*, Journal of Quantitative Linguistics **2** (1995), no. 3, 217–237.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, Springer, 1993.

[10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes, *Press: Predictive elastic resource scaling for cloud systems*, Network and Service Management (CNSM), 2010 International Conference on, IEEE, 2010, pp. 9–16.

[11] Carl Hewitt, *Viewing control structures as patterns of passing messages*, Artificial intelligence **8** (1977), no. 3, 323–364.

[12] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica, *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.*, NSDI, vol. 11, 2011, pp. 22–22.

[13] Jez Humble and David Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*, Pearson Education, 2010.

[14] Philipp K. Janert, *Feedback control for computer systems*, O'Reilly Media, 2013.

[15] Jyh-Shing Roger Jang and Shiuan-Sung Lin, *Optimization of viterbi beam search in speech recognition*, International Symposium on Chinese Spoken Language Processing, 2002.

[16] Frederick Jelinek, *Interpolated estimation of markov source parameters from sparse data*, Pattern recognition in practice (1980).

[17] Frederick Jelinek, *Statistical methods for speech recognition*, MIT press, 1997.

[18] Filip Jurčíček, Ondřej Dušek, Ondřej Plátek, and Lukáš Žilka, *Alex: A Statistical Dialogue Systems Framework*, Text, Speech and Dialogue, Springer, 2014, pp. 587–594.

[19] Reinhard Kneser and Hermann Ney, *Improved backing-off for m-gram language modeling*, Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on, vol. 1, IEEE, 1995, pp. 181–184.

[20] Matěj Korvas, Ondřej Plátek, Ondřej Dušek, Lukáš Žilka, and Filip Jurčíček, *Vystadial 2013–Czech data*, (2014).

[21] Akinobu Lee, Tatsuya Kawahara, and Kiyohiro Shikano, *Julius—an open source real-time large vocabulary recognition engine*, (2001).

[22] K-F Lee, H-W Hon, and Raj Reddy, *An overview of the SPHINX speech recognition system*, Acoustics, Speech and Signal Processing, IEEE Transactions on **38** (1990), no. 1, 35–45.

[23] Richard P Lippmann, *Review of neural networks for speech recognition*, Neural computation **1** (1989), no. 1, 1–38.

[24] Matthew Marge, Satanjeev Banerjee, and Alexander I Rudnicky, *Using the amazon mechanical turk for transcription of spoken language*, Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on, IEEE, 2010, pp. 5270–5273.

[25] Dirk Merkel, *Docker: lightweight linux containers for consistent development and deployment*, Linux Journal **2014** (2014), no. 239, 2.

[26] Nima Mesgarani, Malcolm Slaney, and Shihab A Shamma, *Discrimination of speech from nonspeech based on multiscale spectro-temporal modulations*, Audio, Speech, and Language Processing, IEEE Transactions on **14** (2006), no. 3, 920–930.

[27] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur, *Recurrent neural network based language model.*, INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010, 2010, pp. 1045–1048.

[28] Tim Ng, Bing Zhang, Long Nguyen, Spyros Matsoukas, Xinhui Zhou, Nima Mesgarani, Karel Veselỳ, and Pavel Matejka, *Developing a speech activity detection system for the darpa rats program.*, INTERSPEECH, 2012.

[29] Scott Novotney and Chris Callison-Burch, *Cheap, fast and good enough: Automatic speech recognition with non-expert transcription*, Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Association for Computational Linguistics, 2010, pp. 207–215.

[30] Ondřej Plátek and Filip Jurčíček, *Free on-line speech recogniser based on Kaldi ASR toolkit producing word posterior lattices*, 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue, 2014, p. 108.

[31] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukáš Burget, Ondřej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlíček, Yanmin Qian, Petr Schwarz, et al., *The Kaldi speech recognition toolkit*, (2011).

[32] Lawrence Rabiner and Stephen E Levinson, *Isolated and connected word recognition–theory and selected applications*, Communications, IEEE Transactions on **29** (1981), no. 5, 621–659.

[33] Luis Javier Rodríguez and Inés Torres, *Comparative study of the baum-welch and viterbi training algorithms applied to read and spontaneous speech recognition*, Pattern Recognition and Image Analysis, Springer, 2003, pp. 847–857.

[34] Neville Ryant, Mark Liberman, and Jiahong Yuan, *Speech activity detection on youtube using deep neural networks.*, INTERSPEECH, 2013, pp. 728–731.

[35] David Rybach, Christian Gollan, Georg Heigold, Björn Hoffmeister, Jonas Lööf, Ralf Schlüter, and Hermann Ney, *The RWTH Aachen University Open Source Speech Recognition System*, Interspeech, 2009, pp. 2111–2114.

[36] Matthias Sperber, Graham Neubig, Satoshi Nakamura, and Alex Waibel, *On-the-Fly User Modeling for Cost-Sensitive Correction of Speech Transcripts*, Spoken Language Technology Workshop (SLT), 2014.

[37] Lloyd R Welch, *Hidden markov models and the baum-welch algorithm*, IEEE Information Theory Society Newsletter **53** (2003), no. 4, 10–13.

[38] Jason D Williams, I Dan Melamed, Tirso Alonso, Barbara Hollister, and Jay Wilpon, *Crowd-sourcing for difficult transcription of speech*, Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on, IEEE, 2011, pp. 535–540.

[39] Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al., *The HTK book*, vol. 2, Entropic Cambridge Research Laboratory Cambridge, 1997.

# List of Abbreviations

**API**     Application Programming Interface

**ASR**     Automatic Speech Recognition

**DNN**    Deep Neural Network

**HMM**   Hidden Markov Model

**RTF**     Real Time Factor

**SGMM** Subspace Gaussian Mixture Model

**VAD**    Voice Activity Detection

**WER**    Word Error Rate

# A. CD Content