Runar Andersstuen and Christoffer Marcussen

TaleTUC: Automatic Speech Recognition for a Bus Route Information System

Master thesis, spring 2012

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Problem description

The assignment was given by supervisor Björn Gambäck and co-supervisor Rune Sætre, and is a part of the FUIROS-project:

> *FUIROS - Fremtidens ultimate intelligente ruteopplysningssystem.*
>
> *BusTUC[1] is a natural language bus route system for Trondheim. It gives information about scheduled bus route passings, but has no information about the real passing times. This is about to change, because AtB[2] has installed GPS tracking of the buses, giving access to real passing times and delays. Besides, with new smart phones arriving rapidly on the market, there are possibilities for GPS localisation and connections to maps. The project shall take a broad view, and consider all possible advanced concepts, resulting in advanced smart phone applications.*

The task at hand is to research the possibilities for a solution that combines Automatic Speech Recognition (ASR) and *context-awareness* through Case-based reasoning (CBR), for use in a bus route information system context. It is also desirable to implement a proof of concept system, to be used together with TABuss (Marcussen and Eliassen, 2011) and MultiBRIS (Andersstuen and Engell, 2011), which are existing parts of the FUIROS-project. The end goal is to use TABuss as a TaleTUC client, and integrate TaleTUC's functionalities with TABuss' main functionality, which provides real-time updated route suggestions through BusTUC and AtB's real-time system.

The ASR module can build on existing solutions, such as *Buster* (Hartvigsen et al., 2007), or on new technology, like iPhone's SIRI[3], or other suitable libraries and tools. The main requirement for a final implementation is that it is able to recognise Norwegian spoken bus stop names.

The *context-awareness* module is an extension of the work started by Marcussen and Eliassen (2011), and will be designed to optimise TaleTUC.

[1] `http://busstuc.idi.ntnu.no/`

[2] `www.atb.no`

[3] `http://www.apple.com/iphone/features/siri.html`

Abstract

With the constant increase in smartphone sales, integrated sensors have become available to the average user. This allows for mobile applications to utilise the user's context to provide more accurate information. The popularity of smartphones also attract developers to create audio functionalities that have earlier been restricted to calling interfaces. There is an increasing interest for Automatic Speech Recognition (ASR) services aimed at everyday tasks, where Apple's release of SIRI is a good example of a system that has contributed to the gained popularity.

This report describes TaleTUC, a proof of concept system for the domain of bus route information. TaleTUC uses ASR combined with *context-awareness* through Case-based Reasoning (CBR), to recognise spoken bus stop names. It is built on a client-server architecture, where the TABuss (Marcussen and Eliassen, 2011) Android application has been extended to operate as a client. As a TaleTUC client, TABuss uses speech as input to its main query functionality, which provides bus route suggestions through BusTUC and AtB's real-time system.

Three modules have been developed server-side, where one is used for ASR, and the two others are used for *context-awareness*. Testing of the three modules combined showed improved results compared to the ASR module alone, which indicates that *context-awareness* is a suitable technology to combine with ASR.

Sammendrag

Som en følge av det stadig økende salget av smarttelefoner, har integrerte sensorer blitt tilgjengelige for den vanlige bruker. Gjennom disse kan mobile applikasjoner gi mer nøyaktig informasjon, ved å utnytte brukerens kontekst. Smarttelefonenes popularitet gjør også at utviklere kan lage lydfunksjonaliteter som tidligere kun har vært tilgjengelig over faste telefonlinjer. Det er en økende interesse for programmer som tilbyr tale-til-tekst funksjonalitet rettet mot hverdagslige gjøremål, der Apple's SIRI er et godt eksempel på et system som har bidratt til den økende populariteten.

Denne rapporten beskriver TaleTUC, et "proof of concept" system for bussruteinformasjon. TaleTUC bruker tale-til-tekst kombinert med *context-awareness* gjennom Case-based Reasoning (CBR), for å gjenkjenne talte bussholdeplassnavn. Systemet er bygget på en klient-server arkitektur, der utvidelser har blitt laget i Android-applikasjonen TABuss (Marcussen and Eliassen, 2011) for at den kan bli brukt som en TaleTUC-klient. TABuss bruker da tale som input til dens hovedfunksjonalitet, som tilbyr ruteforslag gjennom BussTUC og AtBs sanntidssystem.

Tre moduler har blitt laget på server-siden. Én er brukt til tekst-til-tale, og de to andre er begge brukt til CBR og *context-awareness*. Utførte tester på en kombinasjon av de tre modulene viste forbedrede resultater sammenlignet med kun tale-til-tekst-modulen alene. Dette indikerer at *context-awareness* er en passende teknologi å kombinere tekst-til-tale med .

Preface

This master's thesis describes the study and work done by Runar Andersstuen and Christoffer Marcussen, in partial fulfillment of a Master of Science (MSc) in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The report is a contribution to the FUIROS-project, where earlier work related to TaleTUC has been conducted by Marcussen and Eliassen (2011), and Andersstuen and Engell (2011).

Section 9.2 in the future work chapter is also found in the FUIROS project by Engell (2012). Subsections 9.2.1 and 9.2.3 were originally written by Andersstuen and Engell (2011). These have been modified by Marcussen, and were approved by Andersstuen and Engell. Subsections 9.2.2 and 9.2.4 were originally written by Marcussen and Eliassen (2011). These have also been modified by Marcussen, and were approved by Eliassen.

Runar Andersstuen and Christoffer Marcussen
Trondheim, June 6, 2012

Contents

List of Figures

List of Tables

Acronyms

ANNs Artificial Neural Networks. 15

API Application Programming Interface. 59

ASR Automatic Speech Recognition. i, 5–9, 11, 13–17, 19–24, 27–33, 35–37, 44, 47, 49, 52, 54, 55, 57, 58, 64, 65, 80–83, 87–93, 95–97, 100, 101, 115, 116, 120

BSF Bus Stop Finder. 9, 47, 57, 58, 64, 67, 70, 71, 73, 80, 81, 88, 89, 91, 92, 95, 115, 116, 119, 120

CBR Case-based reasoning. i, ii, 6–9, 31, 35–38, 47, 49, 58, 61–64, 74, 77, 78, 80, 81, 88, 89, 91–93, 95, 97, 101, 115, 120

CMN Cepstral Mean Normalisation. 54, 55

DVM Dalvik Virtual Machine. 27, 44

GPS Global Positioning System. 36, 44, 45, 65, 77, 93

HCI Human Computer Interaction. 17

HMM Hidden Markov Model. 12, 13, 15, 16, 21, 22, 30, 51

HTK Hidden Markov Model Toolkit. 16, 21–24, 27, 30, 50, 95

HTTP Hypertext Transfer Protocol. 17, 70, 83, 92

IDI Department of Computer and Information Science. iii

JVM Java Virtual Machine. 27

MFCCs Mel-frequency cepstral coefficients. 13, 21

MSc Master of Science. iii

NDK Native Development Kit. 44, 57, 70

NFC Near Field Communication. 99

NTNU Norwegian University of Science and Technology. iii, 29, 30

OOV Out-of-vocabulary. 16, 19, 57, 58, 88, 93

RFID Radio Frequency Identification. 99, 100

SDK Software Development Kit. 44

SLMs Stochastic Language Models. 11

VAD Voice Activity Detection. 53, 55, 72, 73, 96

WAcc Word Accuracy. 14

WAR Web Application Archive. 120

WER Word Error Rate. 14, 19, 28, 30–32, 51, 55, 56, 71, 73, 74, 80, 81, 87–89, 91, 93, 95, 116

Chapter 1

Introduction and Overview

These introduction sections first describe the background and motivation for TaleTUC. The goals and research questions are defined, and the research method is described. Finally, an overview of the whole report is provided.

1.1 Background and Motivation

One of the main research foundations for the work is the existing ASR system *Buster* (Hartvigsen et al., 2007). *Buster* is a spoken dialogue system which interacts with BusTUC, and provides route suggestions through a calling interface. *Buster* is further described in section 2.6.2.

The work is also motivated by the previous FUIROS-projects conducted by Marcussen and Eliassen (2011), and Andersstuen and Engell (2011). These projects explored the possiblities with mobile bus route information systems, and natural language through BusTUC. Reviews of Marcussen and Eliassen's developed application gave indications that it would be a popular choice for bus travellers, which is a motivation to conduct future research and development.

This report presents the development of TaleTUC, which consists of three modules: the Automatic Speech Recognition (ASR) module, the Bus Stop Finder (BSF) module and the Case-based Reasoning (CBR) module. TaleTUC uses a client-server architecture, where a client prototype has been implemented for Android[1] devices. The development is performed to further explore new possibilities within the bus route information domain, and to further utilise smartphone capabilities.

[1] `http://www.android.com/`

1.2 Goals and Research Questions

Research question 1 Which ASR technologies are well suited for Android devices and the task at hand?

Research question 2 Where is it most desirable to do the different parts of the ASR? On the device or on a server?

Research question 3 Can *context-awareness* through CBR optimise the performance of an ASR system?

Goal 1 Develop a prototype system based on the results from research questions 1 and 2.
The prototype system should be designed to recognise 50 bus stop names in Trondheim, and its vocabulary should be easy to expand. The prototype system should also be designed modular, so that it is possible to add additional bus stop names, and also extend to other domains than bus route information.

Goal 2 Develop modules for *context-awareness* based on research question 3, and integrate them with the prototype.
Testing the combination of ASR and *context-awareness* should give answers to whether or not *context-awareness* can optimise the performance of the ASR.

Goal 3 Integrate the prototype with existing parts of the FUIROS-project.
The prototype system should be integrated with the existing Android application TABuss (Marcussen and Eliassen, 2011), where TABuss then will operate as a TaleTUC client. It will be a part of TABuss' main functionality, where only a destination input is needed for the system to provide route suggestions.

1.3 Research Method

Table 1.1 shows the roughly outlined progress plan. A more detailed plan was designed using Trello[2], where a digital board displays tasks (similar to a Scrum[3]-board).

[2] `www.trello.com`
[3] `http://www.scrum.org/`

Date	Milestone description
01.02.2012	Initial research finished
01.03.2012	Point of no return for technology choice
01.05.2012	Prototype finished
15.05.2012	Final draft of report finished
06.06.2012	Report and other deliverables ready for delivery
08.06.2012	Delivery

Table 1.1: Progress plan

The work done followed the following steps:

Problem definition. In this step the problem was defined, and the problem description formulated. It was decided that a large amount of time was needed for research, as the project members had no previous experience with the domain of ASR.

Research. The research used Google Scholar[4] and the digital libraries listed in Table 1.2. For the ASR research, the key search words were:

- Mobile devices
- Architecture
- Route information
- Android
- Engines
- Feature extraction
- Noise

For the CBR research, the key search words were:

- ASR
- *context-awareness*

For the *context-awareness* research, the key search words were:

- ASR
- Route information systems

[4]http://scholar.google.com

- Smartphones

The research was performed in the following stages:

- Research existing ASR engines, existing ASR solutions and combinations of ASR and *context-awareness*. Further research narrowed in on solutions for mobile devices and the route information domain.
- Research combinations of ASR and *context-awareness* through CBR
- Research noise cancellation and filters in ASR
- Research the Android technologies necessary for TABuss to operate as a TaleTUC client, and also other client possibilities

Prototype implementation. This step involved the following stages:

- Develop an ASR module
- Develop a CBR module
- Develop a module for connecting the ASR and CBR modules
- Develop a client

Testing. The tests were performed on the modules individually, and on a combination of the three. Three test sets were used for testing the ASR. Two of the test sets each contained 50 audio files, and the third contained a combination of these (25 from each). Case bases for these audio files were created for testing the CBR.

Thesis writing. This process was continuous. However, most of the writing was done in two time periods. The first was in the research phase, and the second was when the results of the prototype system were documented.

Source	**URL**
IEEE Xplore	http://ieeexplore.ieee.org/Xplore/guesthome.jsp
ACM Digital Library	http://dl.acm.org/dl.cfm
Springer Link	http://www.springerlink.com/?MUD=MP
CiteSeerX	http://citeseerx.ist.psu.edu/
ScienceDirect	http://www.sciencedirect.com/

Table 1.2: Digital libraries used

1.4 Thesis Structure

Chapter 1 contains the introductory sections, which describe: the background and motivation of this thesis, the defined goals and research questions, the research method used and the thesis structure.

Chapter 2 presents the background ASR theory, and the researched ASR technologies.

Chapter 3 presents the background CBR theory, and the researched CBR technologies.

Chapter 4 presents the existing route information technologies.

Chapter 5 provides information on smartphones.

Chapter 6 describes the development of TaleTUC. This involves the development of the ASR module, the Bus Stop Finder (BSF) module and the CBR module, the combination of these and the prototype client.

Chapter 7 presents the development results, with figures, descriptions and screenshots.

Chapter 8 first discusses the development results. This chapter then provides a review of the research questions and goals, before a conclusion is presented.

Chapter 9 provides suggestions for future work.

Chapter 10 provides the acknowledgements.

Chapter 2

Automatic Speech Recognition (ASR)

The following sections first give a description of the key concepts in ASR, and the ASR architecture, before a summary of the developments in the field of ASR is provided. Finally, ASR for mobile devices is described.

2.1 Key Concepts and ASR Architecture

The architecture of an ASR system can be viewed in figure 2.1. O is a given observation sequence, and W is the most probable sentence. $P(W)$ is the *prior probability*, which is computed by the *language model*, and $P(O \mid W)$ is the *observation likelihood*, which is computed by the *acoustic model* (Jurafsky and Martin, 2008). $P(W \mid O)$ is the *prior probability* given an *observation likelihood*, which is computed by the *decoder*.

The language model $P(W)$ is used to restrict a word search, by defining which words can succeed already recognised words. A popular choice for designing *language models* is n-gram models[1]. These are *Stochastic Language Models (SLMs)* aimed at providing adequate probabilistic information, so that likely word sequences should have higher probability (Huang et al., 2001).

For general-purpose speech recognition, the language model can be an n-gram model of text learned from a corpus of written sentences. (Russell and Norvig, 2009). An n-gram model can also consist of transcripts of spoken language, which is more accurate, and takes into account the differences between spoken and written language.

[1] http://www.w3.org/TR/ngram-spec/

In *n*-gram models, we assume that:

$$P(w_i|w_1, w_2, ..., w_{i-1}) \tag{2.1}$$

only depends on the last *n-1* words, which results in:

$$P(w_i|w_{1-(n-1)}, w_{2-(n-1)}, ..., w_{i-1}) \tag{2.2}$$

From this we can also infer the expressions: *unigram*, *bigram* and *trigram* models, based on the number of words used. This leads to:

Unigram model:

$$P(w_i) \tag{2.3}$$

Bigram model:

$$P(w_i|w_{n-1}) \tag{2.4}$$

Trigram model:

$$P(w_i|w_{n-2}, w_{n-1}) \tag{2.5}$$

The acoustic model *P(O | W)* is a statistical representation of transcribed sound, and is where the likelihood of the observed spectral feature vectors given linguistic units is computed. *Acoustic models* play an important role in minimising the error rates, because they are used together with the *language model* in the maximisation of the posterior probabiity *P(W | O)*. It should take into account several factors: speaker variations, pronunciation variations, environmental variations and context-dependent phonetic coarticulation variations. Recording of the spoken utterances for training the *acoustic model* should have the same formats (i.e., sample rate and bits per sample), as the spoken utterances to be recognised. This also means that if the system is designed to receive input through a calling interface, training data should be recorded in the same manner[2].

To train an *acoustic model*, statistical representations of the sounds that make up a word are created for each phoneme. Such a representation is in most cases a Hidden Markov Model (HMM). The training of an *acoustic model* is known as the *learning problem* in HMMs. The solution to this problem is often the *Baum-Welch* algorithm (Baum et al., 1970), which is the traditional method used to train HMMs.Each phoneme has its own HMM. Similar HMM states can be clustered into what is called a *senone*.

[2]http://cmusphinx.sourceforge.net/wiki/tutorialam

Feature extraction is the process of extracting vectors of features from frames, where frames are summarised signal properties over time slices. The typical size of a time slice is 10 ms. The most common features in ASR are Mel-frequency cepstral coefficients (MFCCs), which are frequency-domain features. Computation steps have been described by among others Muda et al. (2010) and Molau et al. (2001), where the use of *Fourier transformations*[3] is the key, in order to express the input signal in the frequency domain. The next steps include approximating the frequency of the human ear, using the *Mel scale* (Pedersen, 1965), before performing a *discrete cosine transform* of the log Mel spectrum. The amplitudes of the resulting spectrum are what are called MFCCs.

The decoder is where the *acoustic model* is combined with the *language model*, for finding the most likely sequence of words. This step originates from Bayes' rule, where *P(O)*, i.e., the probability of a given observation sequence, is the same for all *P(W)*:

$$P(W|O) = P(W)\frac{P(O|W)}{P(O)} \tag{2.6}$$

$$P(W|O) \implies max\, P(W|O) \quad , and \tag{2.7}$$

$$max\, P(W|O) = max\, P(O|W)P(W) \tag{2.8}$$

For decoding, the *Viterbi* algorithm (Forney, 1973) is often used. This algorithm can be viewed as a modified *forward* algorithm (Rabiner, 1989), where the *forward* algorithm represents the *evaluation problem* in HMMs. Instead of summing the probabilities from different paths coming to the same destination state, the *Viterbi* algorithm selects and remembers the best one. This is done because the *forward* algorithm does not provide a state sequence. The probability of the best path is defined by:

$$V_t(i) = P(X_1^t, S_1^{t-1}, s_t = i|\Phi) \quad , \tag{2.9}$$

where $V_t(i)$ is the probability of the most likely state sequence at time t, that has generated the observation X_1^t, and ends in state i. The decoding step with the use of the *Viterbi* algorithm is known as the *decoding problem* in HMMs.

[3] http://mathworld.wolfram.com/FourierTransform.html

Error measuring the performance of ASR systems often involves viewing the Word Error Rate (WER) (Evermann, 1999), which is computed by:

$$WER = 100\% \times \frac{S + D + I}{N} \quad , \tag{2.10}$$

where S is the number of substitutions, D is the number of deletions, I is the number of insertions and N is the number of words in the reference sentence. S, D and I are further described in the listing below:

Substitution - When an incorrect word is substituted for a correct one.

Deletion - When a correct word is removed from the recognised sentence.

Insertion - When an uncorrect word has been added to the recognised sentence.

From the WER the Word Accuracy (WAcc) can be calculated, which is another metric of measuring the performance of ASR systems:

$$WAcc = 1 - WER \tag{2.11}$$

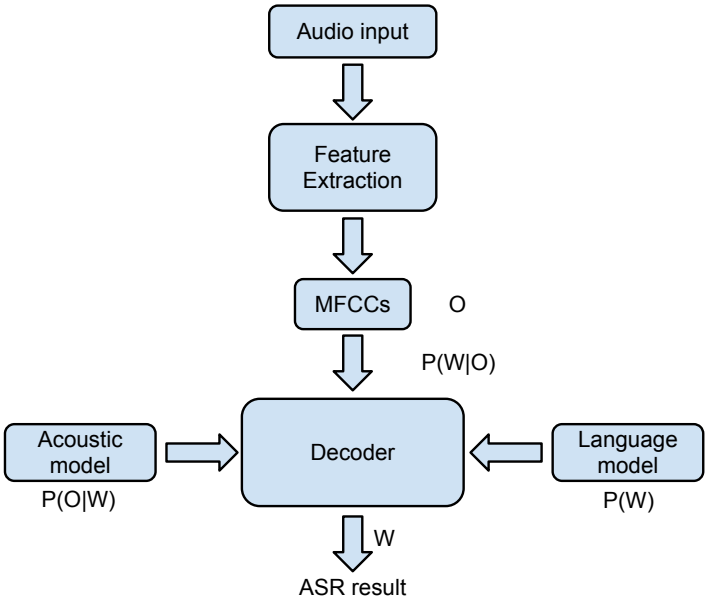

Figure 2.1: ASR architecture

2.2 ASR Development

ASR research was conducted as early as in 1939, when Dudley (1939) of AT&T's Bell Labs[4] proposed the speech analysis model *The Vocoder* (Dudley, 1939). Much of this work was funded and developed for military work[5], and was therefore not meant for the public.

Later, in the 1970s, the ASR technology went through a major development. A basic idea of pattern recognition technology for ASR was proposed by Itakura (1975) and Rabiner et al. (1979). This technology was based on Linear Predictive Coding (Deng and O'Shaughnessy, 2003), which many today may associate with audio codecs.

DARPA[6] established a research program for speech recognition in 1971[7]. This program was aimed at the development of a system that could understand continuos speech (speech with connected words). One of the development results came from the project group at Carnegie Mellon University[8], where Lowerre (1976) published a thesis on the *Harpy* system. The *Harpy* system used graph searching on a sequence of words or sounds, and returned the best reference match. It successfully recognised 1,011 words. During this time period another siginificant system also emerged, which was *Dragon* (Baker, 1975). Baker used Markov processes (Rabiner, 1989) as a part of the ASR, and achieved promising test results, despite a high error rate.

Research performed in the 1980s introduced the *n*-gram model for word searches. Another research area was speaker independence and dialects. Wilpon et al. (1982) published the paper *Speaker-independent isolated word recognition using a 129-word airline vocabulary*, where their system was designed to cope with different speakers, without the need for individual speaker training.

Artificial Neural Networks (ANNs) (Floreano and Mattiussi, 2008) were according to Juang and Rabiner (2005) reintroduced to ASR during the late 1980s. This technology however, was not able to cope with temporal variation, that is, changes over time. On-going ANNs research for ASR did therefore not focus on stand-alone ANNs solutions, but rather combinations of ANNs and HMM (Rabiner, 1989).

The 1980s also saw the foundation of companies such as Dragon Systems and SpeechWorks, which later was aquired by Nuance[9] (see section 2.6.1).

[4] http://www.alcatel-lucent.com/wps/portal/BellLabs

[5] http://www.dragon-medical-transcription.com/historyspeechrecognition.html

[6] http://www.darpa.mil/

[7] http://www.dragon-medical-transcription.com/historyspeechrecognitiontimeline.html

[8] http://www.cmu.edu/index.shtml

[9] www.nuance.com

For the 1990s, Juang and Rabiner states the shift from Bayesian pattern recognition (Box and Tiao, 1992), to error-minimisation, as a milestone. This was addressed by Juang et al. (1997), where they stated that the traditional Bayesian pattern recognition was suboptimal, when the real signal distribution form is not known precisely, as with distribution functions for speech signals. Their argument was that the goal of ASR should be to achieve least recognition error, which is more important than following Bayes criterion.

The *How May I Help You* system (Gorin et al., 1996) was published during this decade, and provided call-routing functionalities through spoken dialogue. For the ASR, a bigram *language model* was used, and a vocabulary of 10 000 utterances. This system also took into account speaker variations and Out-of-vocabulary (OOV) words.

Of the new software that was created during this time period, *Sphinx* (Le, 1988) should be mentioned, which showed remarkable results for large-vocabulary continuos ASRs. *Sphinx*, which is further described in section 2.6.1, combined HMMs with the graph searching idea from the *Harpy* system.

The *Hidden Markov Model Toolkit (HTK)*, which is described in section 2.6.1, is another system developed in the 1990s. The background for its creation was the progress in creating ASR systems with large vocabularies, and the need for research software that could aid the future development. *HTK* is still today a popular ASR software, and was used in the existing *Buster* (Hartvigsen et al., 2007) system.

In more recent years ASR systems have been developed for several domains. An example is *Jupiter* (Zue et al., 2000), which provides weather information through a calling interface. Other systems such as *TravelMan* (Turunen et al., 2007) and *Let's go* (Raux et al., 2003, 2005) focus on transportation services.

There are also a number of companies producing commercial ASR software, where the most known is Nuance. Such systems are often expensive, which makes the access to development tools difficult.

This section has provided a high-level summary of the development in the field of ASR. What is found most interesting and important for TaleTUC are ASR engines such as *Sphinx* and *HTK*, and their contributions to the development of ASRs. These engines are open-source and well documented, which helps the development process. It is also interesting to view the domains that ASR functionality has been implemented in, and the technologies used.

2.3 ASR for Mobile Devices

ASR systems for mobile devices have become popular within the field of pervasive computing (Alewine et al., 2004), and span from the traditional calling

interface to systems developed specifically for smartphones. Raux et al. (2003, 2005) and Hartvigsen et al. (2007) both developed ASR systems with calling interfaces, while Turunen et al. (2007) proposed a smartphone solution. Several technology companies have adapted to the new smartphone trend, and today major companies such as Nuance (see section 2.6.1) and Apple[10] provide ASR solutions for smartphones.

Which architecture to choose for the developemt of ASR systems for mobile devices is a much discussed topic. For applications involving a calling interface there are no architectural choices on where the ASR functionalities should reside, but for the smartphone applications there are two standard architectures to choose between: client-side ASR and server-side ASR (Zaykovskiy, 2006). Client-side ASR benefits from not having to send data to a server. And since it is not dependent on a server, the system will not fail if a server is down.

The other option is a client-server architecture, which is characterised as a modular solution where a server hosts functionalities to several clients. Such clients can be based on any platform, as long as they support transfers over the Hypertext Transfer Protocol (HTTP). This means that both Android and iPhone clients can access the same functionalities, and developers do not have to implement them for each platform. Server-side ASR allows seamless updates of functionalities, as the user never uses these locally. Several projects have embraced this mindset. TABuss' (Marcussen and Eliassen, 2011) bus route information functionalities were shifted to the MultiBRIS server (Andersstuen and Engell, 2011) to save resources and reduce the amount of data traffic. This view is also supported by Turunen et al. (2006a,b, 2007), who designed a distributed architecture with hardware constraints and modularity in mind. Another supporter is the company *MyCaption*[11], which produces voice-enabled applications for smartphones. They have listed server computations as one of the main components in a successful ASR system. Apple's system SIRI also uses a server-side solution, according to actors who have studied the patent application from Apple's R&D labs[12].

The ASR architecture for mobile devices is further discussed in section 2.5, where a view on the challenges of choosing an ASR architecture is provided.

[10] `http://www.apple.com/`

[11] `http://www.mycaption.com/resources/technology/voice-recognition`

[12] `http://www.unwiredview.com/2011/10/12/how-siri-on-iphone-4s-works-and-why-it's-a-big-deal-apple's-ai-tech-details-in-230-pages-of-patent-app/`

2.4 Multimodal Interaction

Multimodal interfaces target multiple ways of interacting with a system, and are key research areas in Human Computer Interaction (HCI) (Dumas et al., 2009). This can be through speech, gestures or other modalities. The advantage of multimodal interaction is that it offers a greater range of communication by adding expressive power. It is adapted towards the user's communicative abilities, and provide improved support for the user's different interaction styles. According to Dumas et al. (2009), 95 % - 100 % of the users preferred multimodal interaction over unimodal input. This topic also has a cognitive foundation. Dumas et al. (2009) states that the user can extract a higher lexical intelligibility by processing multimodal information, and lists the following:

- Humans are able to process modalities partially independently and, thus, presenting information with multiple modalities increases human working memory
- Humans tend to reproduce interpersonal interaction patterns during multimodal interaction with a system
- Human performance is improved when interacting multimodally due to the way human perception, communication, and memory function

Mobile phones have enabled *ubiquitous* spoken telecommunication between humans (Turunen et al., 2007). In later years, mobile devices have become more powerful, enabling the possibility to run multimodal applications. Because they are pervasive, smartphones and multimodality is an appropriate combination. An example is Turunen et al. (2007)'s *TravelMan*, which offers multimodal interaction through speech, haptics, text, physical browsing and physical gestures. Another example is Kühnel et al. (2010)'s *Smart-Home System*, which is a system that can control living room items. These are items such as the TV, the video recorder, the radio and the lamps. This system has a smartphone interface that uses both touch input and gesture recognition.

Multimodal interation in TaleTUC concerns the TaleTUC client, which in this project is TABuss, and the speech synthesis part of TaleTUC (Engell, 2012). The goal for the TaleTUC client is to integrate speech modality with the existing text input. The combination of speech with other modalities was researched alredy in 1980. Bolt (1980)'s *Put-That-There* combined speech with gestures, and commanded simple graphical shapes. Later research by Bangalore and Johnston (2000) focused on the development of language processing techniques of utterances inputted through multiple modalities.

Ehlen and Johnston (2012) created a system for local business information. The interesting result from their research is that users do not always wish to

issue commands multimodally, even though they prefer to have the option to do so. Ehlen and Johnston (2012) stated that this could have something to do with smartphone users being too busy too learn the multimodal commands. Instead, the users preferred the commands that felt natural to them. Ehlen and Johnston (2012) mentions that their map gesture feature was received poorly, because the users expected their devices to determine their locations for them. In TABuss, the application automatically retrieves the user's location, and uses this in queries. Based on Ehlen and Johnston (2012)'s results, the TaleTUC client should therefore continue the same line of thought as TABuss. It is familiar to the existing TABuss users, and would simplify the integration of ASR into their usage. It ensures that the speech input does not differ from the text input, and eliminates the extended linguistic complexity of speaking complete sentences (e.g., "*Sentrum*" instead of "I want to travel from *Ila* to *Sentrum*"). This view is supported by Oviatt (1999), who stated that when free to interact multimodally, users selectively eliminate many linguistic complexities.

2.5 Identified ASR Challenges

This section describes the ASR challenges in TaleTUC, where the identified areas are: robustness, architecture and user frustration.

2.5.1 ASR Robustness

One of the challenges for TaleTUC is the different spoken dialects in Trondheim. In addition to the local inhabitants, there are a large number of students living in Trondheim. There are many different spoken dialects which have to be accounted for, when the users refer to bus stop names or areas. One way to approach this is by training robust *language models* and *acoustic models*. This challenge touches an important topic in ASR, OOV words, where the ASR result does not match any utterances defined or recorded in the created models. In TaleTUC, this can typically occur when the audio input from a user that speaks a strong Norwegian dialect leads to a result that does not match any bus stop name in Trondheim. Approaches for handling OOV words have been proposed by among others Bazzi et al. (2000) and Yazgan and Saraclar (2004), who both adapted their *language models* to contain OOV words.

Because TaleTUC is a system developed for the bus route information domain, and typical usage is both outside and inside, noise is a challenge. Deng and Huang (2004) identified noise as one of the key challenges for modern day ASR systems, where the biggest problem is changing environments. Related to TaleTUC, this could be a situation where the user is walking or cars are passing

by. Deng and Huang (2004) names two technique categories for noise compensation: *feature-space compensation* and *model-space compensation*. These techniques were addressed by Miguel et al. (2007), where they created an adaptive *acoustic model* able to capture speaker variabilites, and a normalisation technique for normalising noisy data, obtaining a WER improvement of 83,45%.

2.5.2 Architecture

As discussed in section 2.3, mobile ASR applications are challenged with hardware limitations, which affect architectural decisions. The option to choose between client-side ASR and a service-oriented architecture with server-side ASR was mentioned. A big challenge for a client-side system is optimisation and computation efficiency, caused by hardware limitations compared to stationary devices. It does not represent a modular solution available to multiple platforms, which means that the functionalities have to implemented for each target platform. In addition, the vocabulary cannot be as large as in a server-side ASR, because of storage limitations. For use in an application for mobile devices, it is not given that a user would download a big application. This is the main reason why server-side ASR often is chosen. However, there is a trade-off between data traffic and resources, when it comes to shifting speech recognising functionality to a server. One disadvantage with server-side computations is that it requires data traffic. Also, the challenge on *pervasive computing* identified already in 1994 by Forman and Zahorjan (1994) on the problems with wireless communication is still present today, allthough it is not as strong. With today's WiFi range, WiFi hot-spots, 3G and so on, the problem is to a certain extent avoided. But there is still a challenge when the user moves from network to network, for example from a WiFi network to a 3G network. The user of an application developed with server-side ASR will also not be able to view what he or she said, before a response from the server is returned.

An optimisation to server-side ASR described by Zaykovskiy (2006) is to perform the feature extraction process on the mobile devices. This is called a distributed architecture, which minimises the size of the sound files sent to the server, and will according to Zaykovskiy (2006) replace the standard client-server architecture. This view is also supported by others. Tan et al. (2005) stated that a distributed architecture would aid the development of ASR systems meant to communicate over WiFi, because of less bandwith usage. Delaney et al. (2005) compared the distributed architecture to client-side ASR. They concluded that a distributed architecture was a better choice, after monitoring CPU cycles and energy consumption.

2.5.3 User Frustration

Raux et al. (2005) identified a frustration factor for users when the system gave incorrect recognition results. For TaleTUC, which is designed for smartphones, many incorrect results will lead to unnecessary data traffic (given that the user has not given up). It was therefore important for the developed TaleTUC client to have a plan for what to do if the system after repetitive attempts cannot return the correct results. To allow the user to confirm the ASR result, before a TABuss query is sent, is also important. This avoids unnecessary data traffic, but the whole process with ASR and the TABuss query will take longer time to perform, as the user explicitly has to initiate the TABuss query.

2.6 Existing ASR Technologies

The following sections describe the researched ASR technologies.

2.6.1 Existing ASR Engines and Tools

In the follow sections the researched ASR engines and tools are presented, and TaleTUC's choice of ASR engine is described.

The Hidden Markov Model Toolkit

The *Hidden Markov Model Toolkit (HTK)* is a toolkit for building and manipulating HMMs. The *HTK Book* is available on the website[13] after registration. This book describes how to use all *HTK* components, and also provides general descriptions of ASR related topics.

A *dictionary* in *HTK* contains a sorted list of the required words, together with their pronunciations. A brief example *dictionary* is shown in listing 2.1

```
ACADEMIES           ax k ae dx ax m iy z
ACADEMY             ax k ae dx ax m iy
ACCADEMIA           ae k ax d iy m iy ax
ACCELERATED         ae k s eh l er ey dx ix d
ACCENT              ax k s eh n t
ACCENTS             ae k s eh n t s
ACCENTUATE          ae k s eh n ch uw ey t
ACCENTUATED         ae k s eh n ch ax w ey dx ix d
```

Listing 2.1: Example HTK dictionary, with pronunciations

[13] http://htk.eng.cam.ac.uk/

A *language model* in *HTK* is an *n*-gram model. It is created using a built-in tool called *HLM*, where the input is a file containing the words to train the recogniser for. An *acoustic model* in *HTK* is built in a stepwise manner. First, there is a file preparation step: the audio data is recorded, and converted to MFCCs, before they are used in the training of the *acoustic model*. Word-level transcriptions of the audio are then created, before the built-in tool *HLEd* is used to expand these transcriptions to phone-level transcriptions. After the file preparation step, monophones (single phones) and triphones (three grouped phones) are created, based on the transcriptions. An example of the two is:

```
Word        Monophone
TRANSLATE [TRANSLATE] t r @ n s l e t
Word        Triphone
TRANSLATE [TRANSLATE] t+r t-r+@ r-@+n @-n+s n-s+l s-l+e l-e+t e-t
```

Listing 2.2: Monophones and triphones

The final step is to build the HMMs.

A *decoder* in *HTK* is responsible for the recognition process. The tool used is called *HVite*, and uses *Viterbi-search* with the *language model*, the *dictionary* and the *acoustic model* as input.

If *HTK* is compared with *Sphinx-4*, many of the similar tools are found. *HTK* includes tools for creating *language models* and *acoustic models*, where *Sphinx-4* uses *SphinxTrain*. However, models created in one of them are not directly portable to the other.

There are several systems available that use models created with *HTK*. For TaleTUC two of these have been researched, *Buster* (Hartvigsen et al., 2007), which uses *HTK* models through the *HAPI* (Odell et al., 1999) recogniser, and *Julius*.

HTK was not chosen as TaleTUC's ASR engine because *Sphinx-4* was found to have better documented tools. In addition, *Sphinx-4* uses the Java programming language, which earlier FUIROS-projects also did. This would make the integration of TaleTUC into other parts of FUIROS easier, and benefit from the project participants' experiences. Since it was a goal to create a client for Android, the use of a Java based ASR engine would be beneficial.

Julius

Julius [14] is an open-source engine for continous ASR, first released in 1998. It has been applied to several languages: English, French, Mandarin Chinese, Thai, Estonian, Slovenian and Korean (Lee and Kawahara, 2009). The downloadable

[14] `http://julius.sourceforge.jp/en_index.php`

source contains the needed tools for running recognition from input such as a microphone or audio files. Unlike *Sphinx* (see 2.6.1) it does not have any built-in functionality for building *language models* or *acoustic models*, but models created by tools such as the *HTK* can be imported.

Julius is developed using the C programming language, and can be used in applications in four different ways:

- Embedded as a C library
- Through client-server communication through sockets
- Recognition process controlled by clients/applications
- As a plug-in extension

Julius was not chosen as TaleTUC's ASR engine because of the lack of model building tools. It cannot adopt models created in TaleTUC's chosen model building tool *SphinxTrain*, which meant that since *HTK*'s model building tool was not used, *Julius* was not an option.

Android Speech Input

Android Speech Input is Google's ASR functionality for Android[15] (Ballinger et al., 2010). One of two *language models* can be used: *free form*, which is designed for dictation purposes, and the *web search* model, which is used for shorter phrases. *Android Speech Input* is hosted by Google's servers, and follows the client-server architecture. It supports multiple languages. However, Norwegian is not yet one of these, and *Android Speech Input* was therefore not used in TaleTUC. It would not be an optimal solution even if the Norwegian language was supported, because it does not allow for modifications of the *language models* and *acoustic models*. This would make the recognition of bus stop names difficult, as it is unlikely that they are in *Android Speech Input*'s vocabulary.

ISpeech

ISpeech[16] is a software that provides both ASR and speech synthesis services, and which uses a cloud-based architecture. Definitions a cloud-based architecture have been proposed by among others Zhang et al. (2010), Vaquero et al. (2009) and Mell and Grance (2011). *ISpeech* provides Software Development Kits (SDKs) to mobile platforms such as Android and iOS[17]. Its basic version

[15] `http://developer.android.com/resources/articles/speech-input.html`
[16] `http://www.ispeech.org/`
[17] `http://www.apple.com/ios/`

is free to download, but the integration of *ISpeech* recognition into an application requires a key. *ISpeech* supports multiple languages, and was used by Sriratanaprapahd and Songwatana (2008) in their implementation of ASR for the Thai language. It also supports the Norwegian language. However, the *language models* and *acoustic models* cannot be adapted towards a specific domain (e.g., bus route information). *ISpeech* was not used in TaleTUC because of this. And, as with Nuance, no architectural choices can be made. One is forced to use the cloud-based architecture.

Nuance and Loquendo

Nuance is a corporation that provides commercial ASR and speech synthesis products. Developers can get access to the Nuance Software Development Kits (SDKs) through the *Dragon SDKs*[18]. The most relevant SDK to TaleTUC is the mobile SDK, which is a cloud-based ASR service able to recognise Norwegian spoken words and sentences. This SDK can be directly used in Android, and is at the present time (May 10th 2012) available through the *NDEV Mobile developer program*, where developers can apply and download it for free.

Loquendo[19] (aquired by Nuance in 2011) is a corporation that provides products similar to Nuance's. The Loquendo ASR software provides a mobile SDK, and a range of additional tools, that let the developers customise the *language models* and adapt the *acoustic models* to their own needs. Trutnev and Rajman (2004) compared Loquendo with other ASR softwares, including Nuance and *HTK*. Their results showed that *HTK*, which is open-source, given a reasonable *acoustic model*, obtained the same results as both Nuance and Loquendo.

There is not much academic information about the ASR technologies in Nuance and Loquendo, because they provide commercial solutions. But it is a known fact that Nuance is one of the market leaders in ASR solutions. It is therefore important to mention, as it represents the state-of-the-art, even though the research and development performed is not open to the public.

Because of the required licenses, the *Dragon Mobile SDK* and the Loquendo mobile SDK were not chosen for TaleTUC. In addition, compared to the use of *Sphinx-4*, *PocketSphinx* and *Julius*, no architectural options are available. The developer has to use the cloud-based solution. The alternative is to purchase Nuance or Loquendo's server solutions.

[18] `http://www.nuance.com/for-developers/`
[19] `http://www.loquendo.com/en/`

Sphinx

Sphinx[20] is a speech recognition software written in Java[21], as a collaboration between the Carnegie Mellon University[22], Sun (now Oracle)[23], Mitsubishi Electric Research Laboratories[24] and Hewlett Packard[25]. The lastest release, *Sphinx-4*, is described by Walker et al. (2004) in the paper *Sphinx-4: A Flexible Open Source Framework for Speech Recognition*.

Figure 2.2 displays the *Sphinx* framework. The primary modules are: the *FrontEnd*, the *Decoder* and the *Linguist*. The *FrontEnd* is a front-end which receives a speech signal from the application, and creates a sequence of *Features*. The Linguist uses its *AcousticModel*, *Dictionary* and *LanguageModel* to create a *SearchGraph*, containing the *language model*, pronunciation information and structural information. The *Decoder* combines the *Features* created by the *FrontEnd* and the *SearchGraph* created by the *Linguist*, to produce a *Result* which is returned to the calling application.

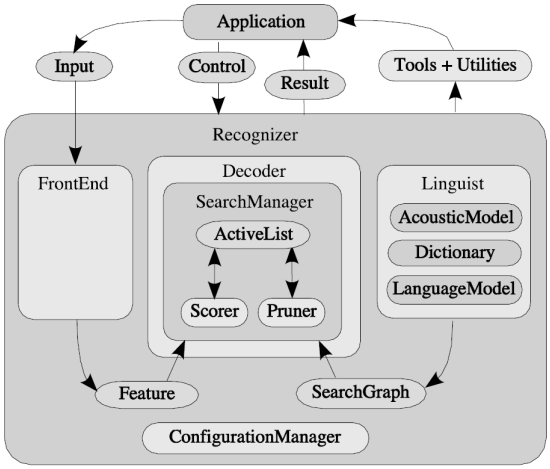

Figure 2.2: The Sphinx-4 framework (Walker et al., 2004)

[20] http://cmusphinx.sourceforge.net/sphinx4/
[21] http://www.oracle.com/us/technologies/java/index.html
[22] http://www.cmu.edu/index.shtml
[23] http://labs.oracle.com/
[24] http://www.merl.com/
[25] http://www.hp.com/

The development in *Sphinx-4* is done with the use of configuration XML-files, where one can set up the different components, including the *FrontEnd* and the *Decoder*. These components are then accessed and used through Java code. An example of a *FrontEnd*-setup in a configuration file is shown in listing 2.3, together with the calling Java code.

```
<component name="epFrontEnd" type="edu.cmu.sphinx.frontend.
   FrontEnd">
        <propertylist name="pipeline">
            <item>audioFileDataSource</item>
            <item>wavWriter</item>
        </propertylist>
</component>

<component name="audioFileDataSource" type="edu.cmu.sphinx.
   frontend.util.AudioFileDataSource">
  <property name="bytesPerRead" value="320"/>
</component>

<component name="wavWriter" type="edu.cmu.sphinx.frontend.util.
   WavWriter">
  <property name="outFilePattern" value="./output_pattern"/>
  <property name="isCompletePath" value="false"/>
  <property name="bitsPerSample" value="16"/>
  <property name="signedData" value="true"/>
  <property name="captureUtterances" value="true"/>
</component>



public class Test {

  public static void main(String[] args) {
    AudioFileDataSource afds = new AudioFileDataSource(320, null);
    WavWriter ww = new WavWriter("./output_pattern", false, 16,
       true, true);
    ww.setPredecessor(afds);

    Data data = null;
    while ((data= ww.getData())instanceof DataEndSignal){
    };
  }
}
```

Listing 2.3: Example frontend configuration

One of *Sphinx*'s strengths is its linguistic modularity, where one can create and use a custom linguistic implementation with a *language model*. A custom

Dictionary can also be used as a pronunciation lexicon, where words are broken down into sequences containing chunks of sub-words. This simplifies the task of implementing a speech recogniser for a non-English language. For TaleTUC, a linguistic implementation would then cover the Norwegian language, and the pronunciation of bus stop names.

In 2006, Huggins-Daines et al. (2006) published a paper on *PocketSphinx*, an open-source licensed version of *Sphinx* for mobile devices. According to Huggins-Daines et al., earlier attempts to implement speech recognisers on mobile devices had suffered from the devices' hardware constraints compared to stationary devices. They therefore aimed to create a non-proprietary toolkit for ASR, optimised for embedded devices, built on *Sphinx-II* (Huang et al., 1992).

Because *PocketSphinx* is not built on the newest version of *Sphinx*, it does not offer the newest *Sphinx* features. Kumar et al. (2011) developed *SphinxTiny*, an adaptation of the 3.X versions of *Sphinx*, for development on mobile devices. Their results showed that *SphinxTiny* performed better than *PocketSphinx* when complex linguistic models were used. However, it was concluded that *PocketSphinx* outperformed *SphinxTiny* when small linguistic models were used. This is an interesting result for TaleTUC. The ASR in TaleTUC would only need to recognise bus stop names (and not complete sentences), and *SphinxTiny* is therefore not automatically a better choice than *PocketSphinx*. Because of this, and because *PocketSphinx* is better documented than *SphinxTiny*, *PocketSphinx* was the preferred version of *Sphinx* among the ones aimed at development for mobile devices.

The biggest difference between *Sphinx-4* and *PocketSphinx* is in where the ASR is performed. *Sphinx-4* is not directly portable to the Android platform, because its libraries are designed to be compiled with the Java Virtual Machine (JVM)[26], and not the Dalvik Virtual Machine (DVM)[27]. Some libraries are as a result not accessible through Android code. If a workaround was available, it would still be a challenge, as *Sphinx-4* is not optimised towards the mobile devices' hardware. A *Sphinx-4* implementation would therefore have to be done server-side. With *PocketSphinx*, an implementation could be done on the device. The difference is then in the data traffic. If *Sphinx* is used, some part of audio has to be transferred and processed on a server, before the recognised text is sent back to the client. No data traffic is necessary in *PocketSphinx*.

For several reasons, *Sphinx-4* became the chosen ASR engine for TaleTUC. Some have already been mentioned in the section on *HTK*. One reason is that the *Sphinx-4* tools and *SphinxTrain* were easy to get started with, and the tutorials were user-friendly. Another reason was on the basis of the architectural

[26] http://java.com/en/download/index.jsp
[27] `http://www.dalvikvm.com/`

challenges discussed in section 2.5.2, where a client-server architecture was the preferred alternative. *PocketSphinx*, and client-side ASR was not chosen because client-side ASR does not represent a modular solution. For TaleTUC, an implementation would then be bound to one platform, and would have to be reimplemented for other platforms. It would still be interesting to develop a small *PocketSphinx* prototype to get a hands on comparison.

SphinxTrain

One of the biggest challenges when making an ASR system for a domain, is to create the required models and *dictionaries*. In order to achieve a low WER, good models are required. The *Linguist* part of the *Sphinx-4* framework requires two models and a *dictionary*, as can be seen in figure 2.2. One of the most time consuming tasks is to create the *acoustic model*, because it requires the collection of training data, in the form of audio files. It also requires a good amount of parameter tuning. *SphinxTrain* is a set of tools that help the user create the *acoustic model* for *Sphinx-4*. A training database needs to be present, in order to train the *acoustic model*. This database contains all the files described in Table 2.1

One of the concerns about *SphinxTrain*, in relation to TaleTUC, was its ability to train *acoustic models* for the Norwegian language. An example of where *SphinxTrain* was successfully used to train a non-English *acoustic model* is described in an article by Satori et al. (2007). This article presents how *SphinxTrain* was used to train a small *acoustic model* for the Arabic language. Satori et al. (2007) achieved high recognition ratios for their tests, which is promising for TaleTUC's training of *acoustic models*.

File	Description
Phonetic dictionary	Contains a phonetic description of all the words to be trained.
Phoneset file	Contains all phones used in the phonetic dictionary.
Language model	Describes a statistical representation of a corpus, that helps the ASR to restrict the word search.
List of filler phones	Contains a list of filler phones. These phones are typical noises like laughter, breath and other phones not described in the language model.
List of files for training	A file that contains the relative paths to all audio files to be used in the training. One file is described per line like this : `speaker_1/file_1`
Transcription	A text file, listing the transcription for each audio file, like this: `<s>helloworld</s>(file_1)`
List of files for testing	A file that contains the relative paths to all audio files to be used in the testing. One file is described per line like this : `speaker_1/file_1`.
Transcription for testing	A text file listing the transcription for each audio file used for testing, like this: `<s>helloworld</s>(file_1)`
Audio files	A set of audio files in the 16khz 16bit mono MS WAV format (Some other formats are supported but not recommended).

Table 2.1: Table of files in a SphinxTrain database.

2.6.2 Existing ASR Solutions

The following sections present the related ASR work, and how this affects the development of TaleTUC.

Buster

Buster is a part of Hartvigsen et al. (2007)'s publication on the *Marvina* system, developed as a part of the *BRAGE*-project[28]. *Buster* is a spoken dialogue system that lets the user pose queries about bus route information through a calling interface. In the *Marvina* system, *Buster* is integrated into a multimodal visitors guide for guests at the NTNU in Trondheim, and is assigned the task on which buses the user has to take to reach the campus. It is a part of *TeleBuster*, which consists of *Buster* and *DAter*. *DAter* is a text-based Directory Assistance system,

[28] `http://www.iet.ntnu.no/projects/brage/`

which provides information on all employees at the NTNU. This is information such as office locations, phone numbers and email-addresses.

The ASR functionality in *Buster* is implemented with the *HAPI* (Odell et al., 1999) recogniser. The models used were created with *HTK*, with a vocabulary size of approximately 700 words. The *acoustic model* was trained with recordings from the *SpeechDat* database[29], made over a fixed telephone line.

Because *Buster* is a program developed by NTNU and Sintef[30], all of the created models are available to the development of the ASR module in TaleTUC. The problem with the use of these is the *acoustic model*, where the *Buster* recordings are not suitable for TaleTUC. These come from a fixed telephone line, and therefore have a sample rate of 8 kHz. For TaleTUC it is desirable to instead use recordings made with a smartphone, with a sample rate of 16 kHz. There is also a difference between how the recordings are done. Often, the microphone is closer to the speaker's mouth when recording over a fixed line, compared to when a smartphone is used. It is, as mentioned in section 2.1, important that the training data is a good representation of what the recogniser is designed to recognise. To use *Buster*'s *acoustic model* would therefore be a suboptimal solution.

CU-Move and CU Communicator

Two systems are presented in the paper *University of Colorado dialog systems for travel and navigation* (Pellom et al., 2001): the *CU Communicator* and the *CU-Move*. The *CU Communicator* is a system similar to *Buster* (Hartvigsen et al., 2007), that integrates ASR, speech synthesis and natural language understanding technologies. It uses a *Sphinx-II* recogniser, for semi-continuous HMMs. The *language model* is a *trigram* model, and is created with the *CMU-Cambridge Statistical Language Modeling Toolkit* (Clarkson and Rosenfeld, 1997). They also created a *dialog context dependent language model*, where different grammars are combined. This improved the WER by 3.5 %.

The *CU-Move* system is a practical implementation of a dialog system, which makes use of the *CU Communicator*. Its main role is to collect audio data. The collection of quality audio data for ASR in an automobile environment can be a challenge. *CU-Move* handles this by applying an array of methods, among those a *Gaussian Mixture Model* based classification of the noise conditions. By integrating an environmental classification system into the microphone array design, they could make decisions on how to best utilise a *noise-adaptive frequency-partitioned iterative enhancement algorithm* (Hansen and Clements, 1991) (Pellom

[29] `http://www.speechdat.org/`
[30] `http://www.sintef.no/`

and Hansen, 1998) or *model-based adaptation algorithms* (Sarikaya and Hansen, 2000) during decoding to optimise the ASR accuracy on the beam-formed signal.

In a later paper called *CU-MOVE: advanced in-vehicle speech systems for route navigation* by Hansen et al. (2005), updated data on the WERs is presented. They achieved a 3.2 % WER for digits, in a noisy environment. To achieve this WER, they used, among other optimisations, something referred to as "Environmental Sniffing". Environmental Sniffing is not filtering of the sound, and not the same as training the *acoustic model* with noise. It uses information about the real-time sound environment to classify the noise conditions, and modify system parameters accordingly. One can therefore claim that the *CU-MOVE* system is *context-aware*. The *context-awareness* in *CU-MOVE* is more integrated in the ASR part of the system, compared to in TaleTUC. Comparisons of the WERs of *CU-MOVE* and TaleTUC can give an indication of how well a loose connection between ASR and *context-awareness* succeeds, compared to a tight connection.

Stopman and Travelman

Turunen et al. (2007) proposed the system *Travelman*, developed for the city of Tampere in Finland. *Travelman* is an updated version of *Stopman* (Turunen et al., 2006b), which provided route planning services.

The ASR in *TravelMan* is used in route searches and in route guidance. It is implemented using a distributed architecture (Turunen and Hakulinen, 2003) and a distributed dialogue model (Salonen et al., 2005), where the mobile devices operate as terminal devices, and a server handles the ASR computations. The usage of terminal devices is something that can be used as an inspiration, and which was introduced to the FUIROS-project by Andersstuen and Engell (2011)'s MultiBRIS project.

An interesting feature in *Travelman* is the use of context and the user's location. The real-time guidance relies on location information, which also can be used to infer departure addresses. The existing application TABuss is *context-aware*, and because one of TaleTUC's goals is to extend TABuss' CBR module, TaleTUC draws inspiration from *TravelMan*.

Let's Go

Raux et al. (2003, 2005) developed a system called *Let's go*, which uses speech through phone calls as input, for returning route information. The goal was to create a grammar model for spoken language, and to include an overall generality regarding the different structuring of sentences with the same meaning.

Let's go was developed using *Sphinx-II* (Huang et al., 1992), and uses the *acoustic model* and the training sets created by Rudnicky et al. (2000) in their dialog system *The Carnegie Mellon Communicator*.

The grammar consists of a 200 000 sentence corpus, which covers most of the bus stops in the Pittsburgh area, and also time expressions. Compared to *Buster* (Hartvigsen et al., 2007), which only maintains approximately 700 names of bus stops and areas, *Let's go* has to maintain information on 15 218 stops. In addition, there are 2423 routes in the Pittsburgh area.

Through their work, Raux et al. (2003, 2005) identified several challenges with speech processing and route information. The main challenge was that different users structured the same phrases differently, when referring to bus stops or places. If *Let's go* is compared to TaleTUC's goals, the biggest difference is the *backends*. While *Let's go* uses a database containing route schedules, TaleTUC uses another approach. In TaleTUC, only a bus stop name is the result of the ASR. This is to be used as input to the existing query functionality in TABuss, where it represents where the user would like to go. TaleTUC then avoids rules on sentence structuring, which is needed in *Let's go*'s speech recogniser, as inserting the ASR result into the correct context is handled by TABuss and MultiBRIS.

Raux et al. (2003, 2005) did not account for different dialects in their training sets. The level of variation in dialects in the Pittsburgh area will not be discussed here, but for Trondheim this is a challenge, which was discussed in section 2.5.

Because *Let's go* uses a calling interface, an important factor is which action the system should perform when the ASR has misinterpreted the user's request. Raux et al. (2005) identified a frustration factor for users during user testing, when the system reprompted a given text caused by misintrepertation. This and other factors such as noise contributed to a WER of 68 %. Another reason why this error rate is so high is the input requirements. The user has to input four pieces of information, which all have to be mapped to the correct context by the system. What is interesting for TaleTUC, is the connection between the user input and the WER. There should be a strive to avoid user frustration, and to minimise the input requirements may be a solution.

SIRI

Apple has not shared much information on SIRI's technology, but some such as the patent application is available. This was briefly mentioned in section 2.3, where some actors have analysed the document and extracted noteworthy information. The most interesting piece of information that can be related to the development of the ASR module in TaleTUC is the use of domains to specify parts of the vocabulary. This can be domains such as: travel, restaurants or

sports. TaleTUC only has one domain, bus route information, but for future ASR work this domain could become one of many domains constituting the ontologies of a FUIROS ASR system.

It has been speculated to be Nuance that has created SIRI's ASR technology, but this has not been confirmed.

As the information released on SIRI does not come from Apple directly, one has to be careful about presenting statements and results. But it is clear that SIRI has become popular among iPhone users, and represent a functional ASR as well as dialogue system. It is therefore worth mentioning, as it represents the state-of-the-art, even though it does not have an academic research documentation.

Chapter 3

Case-Based Reasoning (CBR)

The basis for CBR was allegedly first created by Roger Schank from Yale University in the early 1980s, with Schank's dynamic memory model (Schank, 1983). However, the general interest in CBR first caught on in the early 1990s, when many papers were published and conferences were held. Among the papers published was *Case-based Reasoning: Foundational issues, methodological variations, and system approaches*, by Aamodt and Plaza (1994). This paper gathers terminologies and principles from many former papers, and constructs a formal description of a CBR system. The most notable part of this description is the four processes in the CBR cycle:

1. RETRIEVE the most similar case or cases
2. REUSE the information and knowledge in that case to solve the problem
3. REVISE the proposed solution
4. RETAIN the parts of this experience likely to be useful for future problem solving

A new problem is solved by *retrieving* one or more previously experienced cases, *reusing* the case in one way or another, *revising* the solution based on the *reused* case and *retaining* the new experience by incorporating it into the existing knowledge base. An illustration of this cycle can be seen in figure 3.1.
Case-Based Reasoning(CBR), generally speaking, is the process of solving new problems based on solutions of similar past problems.

3.1 ASR and CBR

Maier and Moore (2009) points out that ASR systems lack procedural knowledge. They argue that the performance of state-of-the-art ASR systems is ap-

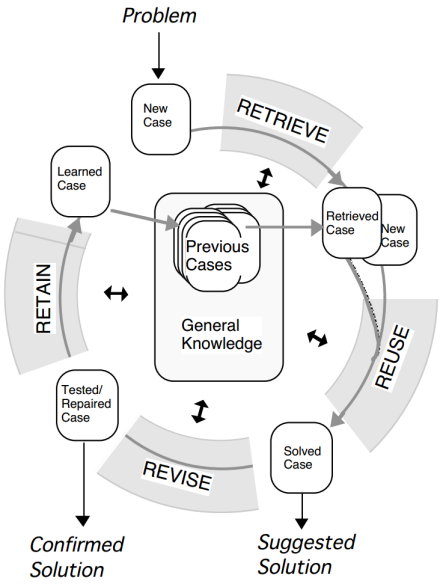

Figure 3.1: The classical representation of the CBR cycle (Aamodt and Plaza, 1994)

proaching an asymptote at a level that falls well short of that which is desirable for many advanced applications (Lippmann, 1997). They further imply that opening up ASR systems to use procedural knowledge through CBR can counteract this asymptotic performance trend. The paper does not present any results from an existing system to back up their theories. However, their argumentation is supported by other research, giving a good indication that combining CBR and ASR could yield good results.

3.2 Context-Awareness and CBR

Smartphones today are pervasive and personal. This means that they are almost always turned on, and they are customised to each user. Raento et al. (2005) claimed that because of this, smartphones are well suited for *context-aware* applications.

Hazas et al. (2004) stated that *context-awareness* is at the core of location-aware computing. Location-aware systems use the user's location through a location-sensing technology such as GPS or WiFi, to provide services.

A system that combines *context-awareness* and CBR was proposed by Ma et al. (2005), who based context on factors such as the environment and user attributes. These factors together with the actions the user performs are stored

as cases.

A *context-aware* bus route information system should be able to monitor the user's behaviour through sensor input, and use this data to provide route suggestions or other information. TaleTUC draws inspiration from TABuss' (Marcussen and Eliassen, 2011) combined usage of *context-awareness* and CBR. The goal for TaleTUC is to extend this functionality, and use it to optimise the ASR results (see section 6.4).

3.3 Identified CBR Challenges

According to Aamodt and Plaza (1994), there are five general CBR challenges: case representation, case retrieval, case reusal, case revision and case retainment. Knowledge contained in a case, often called the case memory, has to have some form of representation. This representation, in addition to containing the knowledge, has to be translated into descriptors for the case. Because the CBR module in TaleTUC needs fairly simple knowledge, it was possible to create descriptors that could also serve as the case memory itself. The knowledge needed to be represented in a case is time, location and the desired destination. One of the description challenges encountered related to the representation of time-of-day, is closely connected to the retrieval method used. The retrieval method does its *initial matching* by finding the n closest cases using the Euclidean distance[1]. A typical representation of time-of-day can be done by dividing up the day in 24 discrete sections, as was done by Marcussen and Eliassen Marcussen and Eliassen (2011) in their CBR module. A problem with this representation is that time-of-day becomes more powerful and imprecise than preferred. The reason for this problem can be illustrated by imagining the following scenario: in the case database there is a case that describes that when located at location A and the time is 12:01, the user wants to go to location C. Another case in the case base describes that when located at A and the time is 11:01 the user want to go to location E. If the time when the system checks the case database is 11:59, the initial matching would return the case where the desired destination is E, even if the time is much closer to 12:01 than 11:01. A benefit of the 24 discrete sections is that computation is saved on the initial match, as all cases outside the discrete time window can be ignored.

Another challenge with representing time is the fact that 01:00 is closer, in the amount of time, to 23:00, than 06:00 is to 01:00, because of the clock's cyclic time scale. However, 01:00 is closer to 06:00 numerical. This can theoretically result in problems around midnight, as cases with time descriptions from be-

[1] `http://mathworld.wolfram.com/Distance.html`

fore midnight would be judged as very far off, when the clock has just passed midnight.

When it comes to representing locations, the most obvious way to do so is to represent the location with latitude and longitude. The matching is done by representing the latitude and longitude values as a vector, and calculating the Euclidean distance, as was done by Marcussen and Eliassen citeMarcussen2011. This approach will produce a CBR matching where movement in the north-south direction has a larger impact on the matching than movement in the east-west direction. This happens because the longitude scale goes from minus 180 degrees to plus 180 degrees, while the latitude scale goes from minus 90 degrees to plus 90 degrees. The size of the errors will vary depending on where on the earth the cases take place, because of the properties of longitude and latitude.

Chapter 4

Underlying Route Information Technologies

This chapter provides information on the relevant existing route information technologies in Trondheim. First, a description of BusTUC is provided. AtB's real-time system is then described, before finally, the FUIROS-projects TABuss and MultiBRIS are presented.

4.1 BusTUC

BusTUC is a natural language bus route system for Trondheim, which provides information about scheduled bus route passings. Query examples are shown in figure 4.1, where two different syntaxes are illustrated.

1. Når går bussen fra Samfundet til Torvtaket?
   When does the bus departure from Samfundet to Torvtaket?
2. (Samfundet +n, Prinsen +n) til Torvtaket.
   English translated version not supported.

Figure 4.1: The BusTUC query syntax.

The n represents walking distance (in this case to Samfundet). This representation is actually not used in BusTUC's parsing, but is syntactically necessary when posing queries. MultiBRIS (Andersstuen and Engell, 2011) uses syntax number two, as this allows the system to specify more than one departure bus

stop. The format of the result from BusTUC can be decided by the developer. It can either be plain text or a JSON[1] object.

4.2 AtB's Real-time System

AtB's real-time system provides information on the actual arrival times of all the buses, and is used by MultiBRIS (see section 4.3). Queries and answers are sent and received using a SOAP[2] interface. The only necessary input parameter is a bus stop's real-time ID. The real-time IDs can be retrieved from AtB's server, which holds a list that maps bus stop IDs to real-time IDs. This list is updated from time to time, and to have an updated list is necessary in order to query the correct real-time data. An example illustrating the bus stop ID to real-time ID mapping, is shown in Table 4.1.

Bus stop ID	Real-time ID
16000001	111111
16000002	111112
16000003	111113
...	...

Table 4.1: Bus stop and Real-time ID mapping. New real-time IDs are assigned when the real-time server restarts.

A query to AtB's real-time system returns the five next bus arrivals for the chosen bus stop. The answer contains route numbers, planned arrival times, actual arrival times and destinations.

[1] http://www.json.org/
[2] http://www.w3.org/TR/

4.3 TABuss and MultiBRIS

TABuss is the existing Android application, developed by Marcussen and Eliassen (2011) as a part of the FUIROS-project. TABuss' main functionality provides route suggestions through a query to the MultiBRIS (Andersstuen and Engell, 2011) server. The MultiBRIS server is one of two parts that constitute the MultiBRIS system. It is also a part of the FUIROS-project, and handles the communication with BusTUC and AtB's real-time system.

As seen in figure 4.2, the first step in TABuss' query functionality is a query to MultiBRIS, containing: a destination string (i.e., where the user wants to travel), the location of the user and addtional parameters. The MultiBRIS server uses the location information in a query to BusTUC, where the closest bus stops to the user's location are set as the departure stops. The received destination string is set as the destination. The returned route suggestions are then updated with real-time departure times, retrieved from a query to AtB's real-time system (see section 4.2). The response returned to TABuss from the MultiBRIS server contains a JSON-object with route suggestions. This is then parsed and presented.

Figure 4.3 illustrates this from the user's perspective. The first screenshot shows the application home screen. From here, the user can either type a destination in the input text field, or select one of the suggestions listed below. The second screenshot displays the query result, where a route suggestion is shown.

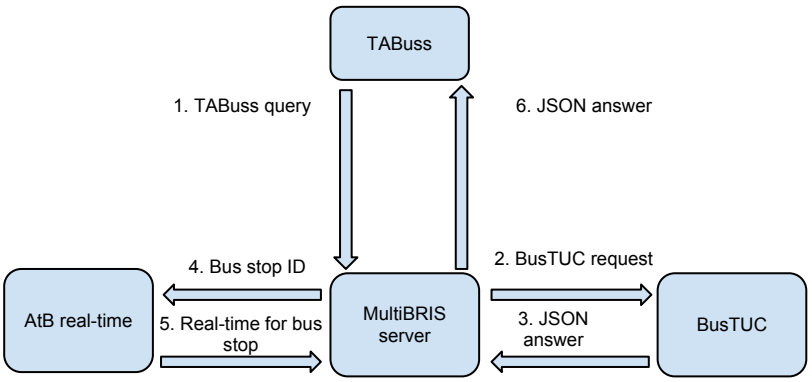

Figure 4.2: TABuss' main functionality

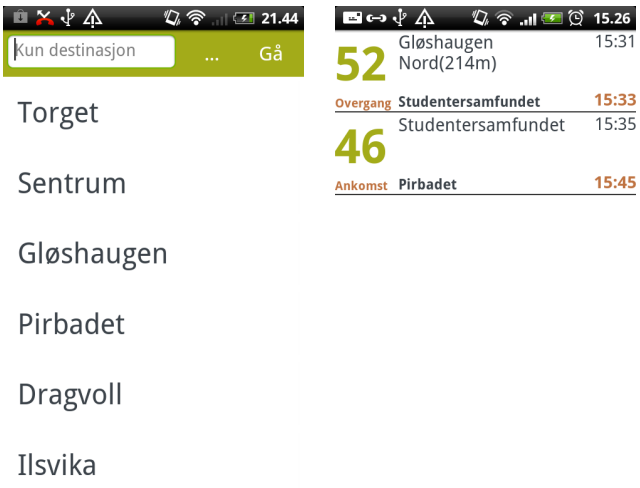

Figure 4.3: Screenshots of the TABuss application

Chapter 5

Smartphones and Android

This chapter first provides updated information on the different smartphone manufacturers' market share. The Android operating system is then described. Finally, the devices that have been used for the development, and their specifications are listed.

5.1 The Popularity of Smartphones

The development and sales of smartphones have expanded immensely over the last few years. Companies such as HTC[1], Samsung[2] and Apple are all focusing on the development of smartphones, where HTC and Samsung use Android as a platform, and Apple iOS. Nokia[3] has also begun to develop smartphones, and has shifted their focus from the Symbian Operating System[4] to the new and emerging Windows Phone 7 platform[5]. According to Gartner's[6] report for the third quarter of 2011, Android had 25,3% of the smartphone market share, while iOS had 16,6%, and Microsoft[7] 2,7 %.

Along with the growth in smartphone sales, the development possibilities increase, and applications targeting every day use have become common. Examples of popular applications are: instant messaging, Facebook[8], e-mail and cardio training.

[1] http://www.htc.com/
[2] http://www.samsung.com/
[3] www.nokia.com
[4] http://www.developer.nokia.com/Community/Wiki/Symbian_OS
[5] http://www.microsoft.com/windowsphone/en-gb/default.aspx
[6] http://www.gartner.com/it/page.jsp?id=1848514
[7] www.microsoft.com
[8] www.facebook.com

5.2 Android

Android[9] is an operating system originally developed by Google and the Open Handset Alliance[10]. Android uses a Linux based kernel containing drivers, with above layers consisting of libraries and the DVM, frameworks and the top application layer.

Android is licensed under Apache[11], and different manufacturers such as HTC and Samsung adapt their own distributions by adding functionalities and a custom user interface. This is similar to how Linux has become available in several different versions like: SuSe, RedHat and Ubuntu.

The Android Software Development Kit (SDK) contains all the necessary classes, packages and files, for developing on the Android platform. The SDK is freeware, and it is available for Windows, Linux and Mac OS. It offers the possibility to target different Android versions, and also provides access methods to device hardware such as GPS, camera and accelerometer. Other features include: media support, database integration and optimised graphics (based on the OpenGL ES framework[12]). TaleTUC targets the SDK-version 2.2 or newer.

For Android development, the Android SDK and Java are the most commonly used tools. C or C++ code can also be integrated through the Android Native Development Kit (NDK)[13] which can be seen as an add-on to the original SDK.

5.3 Devices

Two devices were used in the development of the ASR module, and the client, in TaleTUC: an HTC Desire HD and an HTC Sensation. These were chosen because the department provided the Desire HD, and one of the project members already had the Sensation.

The two devices have similar specifications (see Table 5.1). Both devices are powerful, and can handle heavy computations. An absolute requirement is that an SD-card is present and can be used for storage. This has to do with the storage of audio files created by TaleTUC, and also TABuss' requirements on storage of bus stop information.

[9] `http://www.openhandsetalliance.com/android_overview.html`
[10] `http://www.openhandsetalliance.com/`
[11] `http://www.apache.org/`
[12] `http://www.khronos.org/opengles/`
[13] `http://developer.android.com/sdk/ndk/index.html`

Spec	Desire HD	Sensation
CPU	1 GHz	1,2 GHz
Android version	2.3.4	4.0.3
Read only memory	1.5 GB	1 GB
RAM	768 MB	768 MB
Screen res	800x480	960x540
SD-card size	8 GB	8 GB

Table 5.1: Specifications of the devices. Components such as GPS, WiFi and 3G capabilities are not listed, as these are standard on most smartphones released in later years.

Chapter 6

Method

Figure 6.1 shows the top level view of TaleTUC. The following sections describe the development of: the server, the ASR module, the BSF module and the CBR module.

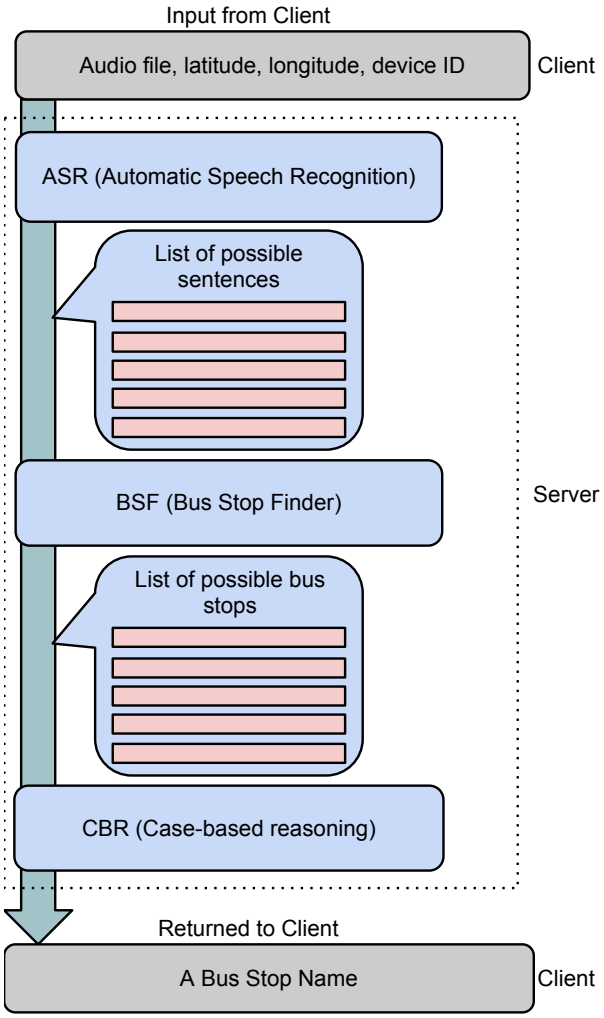

Figure 6.1: Top level view of TaleTUC

6.1 Server

The server technology used is Java Servlet[1]. A Java Servlet can be published through any available servlet container. A servlet container is the web server component that interacts with a servlet, and enables the communication between a servlet and a web-client. TaleTUC, as well as MultiBRIS (Andersstuen and Engell, 2011), use the non-commercial container Jetty[2]. This container is also used by Google, in their Google App Engine[3]. Jetty is developed in Java, which makes it a very portable server solution.

TaleTUC uses three Java Servlets: one for the ASR module , one for the CBR module and one for the Android training application (see section 6.2.1).

6.2 The ASR Module

This section first describes the steps for training the necessary ASR models. Descriptions of the ASR component in figure 6.2 are provided, and measures taken to make the recogniser more robust are presented. Because of the advantages with server-side ASR discussed in section 2.5, TaleTUC uses a client-server approach. It also embraces the distributed architecture mindset, with a created module that allows for cepstrum extraction on the mobile devices.

The numbers in figure 6.2 indicate the order of the performed operations.

[1] `http://www.oracle.com/technetwork/java/javaee/servlet/index.html`
[2] `http://www.eclipse.org/jetty/`
[3] `http://code.google.com/intl/no/appengine/`

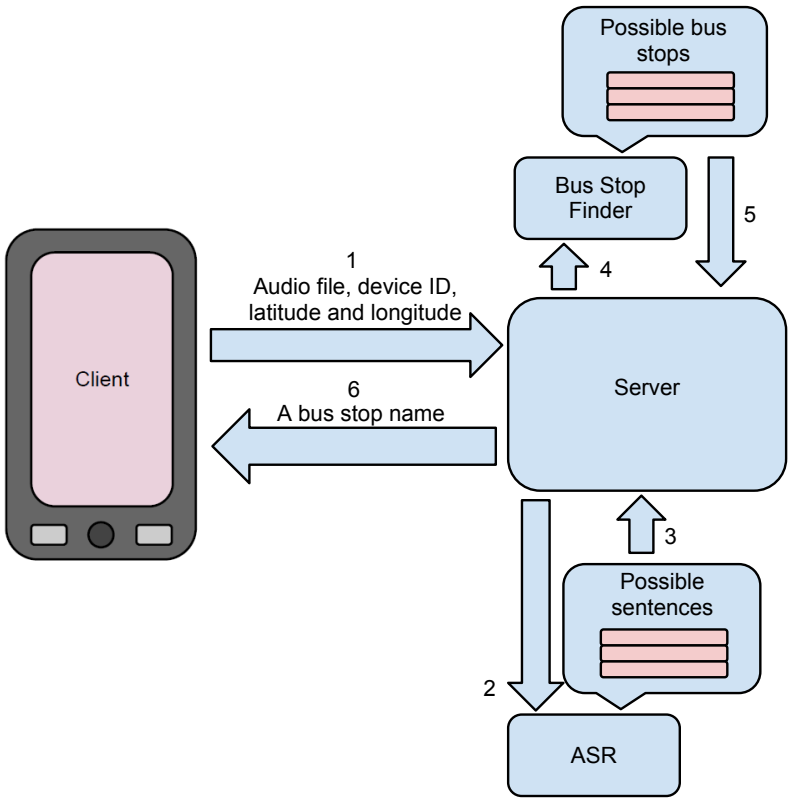

Figure 6.2: TaleTUC ASR flow chart

6.2.1 ASR Models

The *language model* in TaleTUC was created with the *CMU-Cambridge Statistical Language Modeling Toolkit* (Clarkson and Rosenfeld, 1997), and by following the instructions on the *Sphinx* website[4]. A Java program was developed to automate these steps, where the only input needed is a text corpus containing the words, in this case the bus stop names. This Java program runs all the needed scripts, and creates the *language model* files named according to the *Sphinx-4* instructions.

For creating the *acoustic model* in TaleTUC, *SphinxTrain* (see section 2.6.1), was used. 50 of the pronunciations from the *Buster* grammar were mapped to the *Sphinx-4* pronunciation alphabet. Examples of the difference between pronunciations in *Sphinx-4* and *HTK* are shown in Table 6.1. Descriptions of the *Sphinx* phonemes can be seen at `http://www.speech.cs.cmu.edu/cgi-bin/cmudict`. Descriptions of the *HTK* phonemes are found in the *HTK Book* (see

[4] `http://cmusphinx.sourceforge.net/wiki/tutoriallm`

section 2.6.1). The Android devices listed in Table 5.1 were used to record the spoken utterances. An Android application was developed, which first downloads a text file containing the bus stop names. These are displayed in a clickable list, where a click triggers the built-in audio recorder to start recording. When the recording is finished (closed by the user), an audio file is stored on the device's SD-card, with the registered username and the name of the clicked bus stop name as the filename. This file is sent to a server, which stores it in a hierarchy, where files are stored according to the received username and file name from the client. Screenshots of this application are shown in the Result section (section 7.1.1).

Word	***Sphinx-4* pronunciation**	**HTK pronunciation**
Gløshaugen	G L OX S H AE UH E N	g l ox: s h ox uh g e n
Ila	I L A	i: l A
Nova kinosenter	N O V A C I N U S E N T E R	n O: v A C i: n u s e n t e r

Table 6.1: Comparison of the pronunciations in HTK and Sphinx-4

The training of the *acoustic model* followed the steps described in section 2.6.1. The program developed for building the *language model* was extended to also automate the *acoustic model* building steps. It builds the *acoustic model*, and allows the adjustment of configuration parameters. An advantage of using this program is that training can be performed with different parameters sequentially, without having to configure and initiate each run explicitly. It also decodes the trained model, and outputs the WER to a text file, together with the parameter settings the user would like to log. In TaleTUC's case, it was interesting to log the number of *senones* (clustered similar HMM states), and the number of final *densities* (Zhao et al., 1991) used, in order to find the configurations that lead to the lowest WER.

The Java program developed to automate the building steps for the *language model* and the *acoustic model* can currently only be used in Unix-based operating systems. This has to do with Linux being the operating system that was used for the development of TaleTUC, and because the training steps target a Unix environment. For training the models using the Windows operating system, one has to manually perform the training steps from the *Sphinx* website. This can be done with a Unix-like environment such as *Cygwin*[5].

[5] `http://www.cygwin.com/`

6.2.2 ASR Implementation

Two decoders were created for the implementation of the ASR in TaleTUC. One decoder takes a wav-file as input, and the second takes a cepstrum file as input. In total, three *Sphinx-4* front-end configurations were developed. One which is a front-end for the decoding of wav-files (listing 6.1), and another which is a front-end for the decoding of cepstrum files (listing 6.2). The third is used to extract cepstrums, and is accessed by a Java program that extracts cepstrums from a wav-file (listing 6.3), before saving them to a file. The cepstrum front-end uses this file as input, and calculates the delta and double delta (first and second order derivative) of the input cepstrum, before it outputs a feature vector which the decoder uses as input. For more information on the delta and double delta calculations in *Sphinx-4*, the reader is referred to the *Sphinx-4* website on feature extraction[6]. It was chosen to implement two decoders to compare the size of the files (wav and cepstrum) needed to be sent to the server for recognition, and also to compare each solution's time use. The computation of cepstrums was implemented through existing functionalities in the *Sphinx-4* framework, which were ported to the Android platform. The advantage of following *Sphinx-4*'s approach was that all steps from the cepstrum extraction to the recognition process were implemented in the same framework.

```
<component name="epFrontEnd" type="edu.cmu.sphinx.frontend.
   FrontEnd">
        <propertylist name="pipeline">
            <item>audioFileDataSource </item>
      <item>dataBlocker </item>
            <item>speechClassifier </item>
            <item>speechMarker </item>
            <item>nonSpeechDataFilter </item>
            <item>preemphasizer </item>
            <item>windower </item>
            <item>fft </item>
            <item>melFilterBank </item>
            <item>dct </item>
            <item>batchCMN</item>
            <item>featureExtraction </item>
        </propertylist>
    </component>
```

Listing 6.1: Wav decoding front-end

[6] http://cmusphinx.sourceforge.net/sphinx4/javadoc/edu/cmu/sphinx/frontend/feature/DeltasFeatureExtractor.html

```
<component name="epFrontEnd" type="edu.cmu.sphinx.frontend.
   FrontEnd">
        <propertylist name="pipeline">
            <item>streamDataSource</item>
            <item>featureExtractor</item>
        </propertylist>
    </component>
```

Listing 6.2: Cepstrum decoding front-end

```
<component name="mfcFrontEnd" type="edu.cmu.sphinx.frontend.
   FrontEnd">
        <propertylist name="pipeline">
            <item>streamDataSource</item>
      <item>dataBlocker </item>
            <item>speechClassifier</item>
            <item>speechMarker</item>
            <item>nonSpeechDataFilter</item>
            <item>preemphasizer</item>
            <item>windower</item>
            <item>fft</item>
            <item>melFilterBank</item>
            <item>dct</item>
      <item>batchCMN</item>
        </propertylist>
    </component>
```

Listing 6.3: Cepstrum computation front-end

The following descriptions are found in the *Sphinx-4* javadoc[7], and describe the properties in listings 6.1, 6.3 and 6.2.

audioFileDataSource - is the input audio file. This property is used for the decoding of wav files.

streamDataSource - is the input audio stream. This property is used for the decoding of a cepstrum stream.

speechClassifier - is *Sphinx-4*'s implementation of a VAD (Ramirez et al., 2007). The *speechClassifier* holds a user defined threshold, which is used for the processing of incoming audio frames. If the average signal level in decibel is higher than the background noise level by this threshold, the audio frame is marked as speech. A low value indicates that a smaller volume difference is used to separate foreground noise from background noise, which makes the *speechClassifier* more sensitive.

[7] `http://cmusphinx.sourceforge.net/sphinx4/javadoc/`

speechMarker is the processing of an audio file that has been passed through the *speechClassifier*, where regions that are considered to be speech are marked.

nonSpeechDataFilter is the further processing of an audio file that has been passed through the *speechClassifier* and the *speechMarker*, where the non-speech regions are removed.

preemphasizer is the implementation of a high-pass filter, that compensates for the attenuation in the audio data.

windower is where the audio is sliced up into a number of overlapping windows. A *Windowing* function (Rozman and Kodek, 2006) is then applied to each. It is a part of the *Fourier transformation* of the speech signal.

fft is the process of analysing the speech signal into its frequency components, by analysing one window of data.

melFilterBank is where the power spectrum is filtered through a *mel filter bank*, which is a set of triangular filter banks used to approximate the frequency resolution of the human ear.

dct is the *discrete cosine transform* of the input data, after a logarithm has been applied. This is the last step of the conversion of a signal to cepstrum.

batchCMN is *Sphinx-4*'s batch mode implementation of Cepstral Mean Normalisation (CMN) (Liu et al., 1993). This step is included to reduce the distortion caused by the transmission channel.

featureExtractor is where the *delta* and *double delta* of the input cepstrum is computed.

When the ASR process has finished, a word lattice is created. This lattice contains a list of the sentences evaluated throughout the recognition process, sorted based on their score (i.e., the best ASR result is the first lattice sentence).

6.2.3 ASR Robustness

ASR robustness was identified as a challenge for ASR systems in section 2.5. This section describes TaleTUC's approach to cope with this challenge.

Robustness and noise cancellation in ASR systems are much discussed topics, where many have proposed solutions. Sarosi et al. (2011) emphasises the importance of robust features, and compared different feature extraction techniques. Lee et al. (2011) also focused their research on features, and designed a

model for the representation of features, which is able to dynamically adapt to noise.

Narayanan et al. (2011) proposed a solution where multiple prior models are trained instead of only one. These are then used to reconstruct noisy speech. Narayanan et al. (2011) achieved a WER improvement of 13,7 % using this technique.

The first step for TaleTUC was to increase the robustness of the *acoustic model*, and to extract robust features with the use of *Sphinx-4*'s CMN. To make the trained *acoustic model* more robust, the audio for the training sets was recorded in different environments. The recording took place both inside and outside, capturing both an environment where there is little background noise, and an environment where there is traffic noise. Different Norwegian spoken dialects were also taken into account (figure 6.2), as there are many dialects in Norway. This resulted in ten training sets, which were used to train two *acoustic models*. One of the models contains only the training sets with audio recorded indoors, while the other contains the training sets with audio recorded both indoors and outdoors. In addition, three independent test sets were created: one that contains audio recorded indoors (low noise level), one that contains audio recorded outdoors (traffic noise) and one that contains audio recorded both indoors and outdoors. The procedure of training more than one *acoustic model* has previously been done by Cosi and Nicolao (2009). Their results indicated that an *acoustic model* that contains recorded audio with dynamic background noises performs better than an *acoustic model* trained only with homogeneous background noise. For TaleTUC, this also has to do with where the system is to be used. And because the usage area is considered to be both inside and outside, diverse training may be the best alternative.

Different forms of filters for removing noise were investigated. Sound files were mounted in the sound processing program Audacity[8], and different filters were applied. The result was then exported and played in a standard media player, and also used as input to the recogniser. SoX[9] was also used, which has a built-in noise cancellation filter. Based on these experiments, the *Sphinx-4 preemphasizer*, which as mentioned in section 6.2.2 is a high-pass filter, was included.

Sphinx-4's VAD functionality through the *speechClassifier* was also added. Different thresholds were tested, in order to get the best recognition results compared to recognition without filters.

The final step was to include *Sphinx-4*'s *Wiener filter* implementation. The use of a *Wiener filter* in ASR has previously been researched by among others

[8] `http://audacity.sourceforge.net/?lang=nb`

[9] `http://sox.sourceforge.net/`

Arakawa et al. (2006), Chen et al. (2006) and Xu et al. (2005). The filter was added as a property to the front-end in listing 6.3, and combined with the *speechClassifier*'s tuned threshold. A new *acoustic model* was trained on *Wiener filtered* audio, recorded indoors and outdoors. All of the created test sets were tested against this model, in order to view and compare the WER to earlier results with the noisy *acoustic model*. The updated cepstrum computation front-end is shown in listing 6.4.

```
<component name="mfcFrontEnd" type="edu.cmu.sphinx.frontend.
   FrontEnd">
        <propertylist name="pipeline">
            <item>streamDataSource</item>
            <item>dataBlocker</item>
            <item>preemphasizer </item>
            <item>windower </item>
            <item>fft </item>
            <item>wiener </item>
            <item>speechClassifier </item>
            <item>speechMarker </item>
            <item>nonSpeechDataFilter </item>
            <item>melFilterBank </item>
            <item>dct </item>
            <item>batchCMN</item>
        </propertylist>
    </component>
```

Listing 6.4: Cepstrum computation front-end with a Wiener filter

Dialect type	**Number of persons**
North Norwegian	2
East Norwegian	5
North-West Norwegian	1
South Norwegian	1
South-West Norwegian	1

Table 6.2: Overview of trained dialects

6.2.4 Prototype Comparisons of Sphinx-4 and PocketSphinx

A basic *PocketSphinx* prototype was developed for comparison with the *Sphinx-4* solution, where it was interesting to view the differences in storage requirements. Both of the prototypes were tested with the devices listed in Table 5.1. The *PocketSphinx* program required storage of a number of files on the SD-card, such as grammar files, which were manually added during testing. The same *language models* and *acoustic models* were used in both the *Sphinx-4* solution and the *PocketSphinx* solution.

The development of the *PocketSphinx* program was done by following a tutorial on the *Sphinx* website[10]. Existing classes, developed in the C programming language, were imported with the Android NDK, and modified to fit the trained models.

6.3 The Bus Stop Finder (BSF) Module

The BSF, depicted in figure 6.2, is a module that was added server-side, to optimise where the ASR results are OOV words. The BSF uses the word lattice outputted by the ASR as input, and performs the following steps:

1. Checks whether the first lattice sentence matches a bus stop name. If so, this sentence is returned.

2. If the BSF reaches step two, it means that the first lattice sentence does not match a bus stop name, and is an OOV word. The BSF then iterates through the lattice and maps each sentence to the most likely bus stop name, by calculating a new score based on each sentence's similarity to legal bus stop names. This similarity is calculated based on: if the number of words in a lattice sentence matches the number of words in a bus stop name, if any words in the lattice sentence matches any words in a bus stop name and if the matching words have the same placement. It also calculates a confidence score for each bus stop name. The confidence score represents how confident the system is in the bus stop name's score. The bus stop name with the highest calculated score is then returned.

The BSF's scoring uses the following adjustable parameters:

CONF_SCORE_CAP: If the score of a lattice sentence is higher than one, and higher than the CONF_SCORE_CAP, its score is directly added to the score list.

[10] `http://cmusphinx.sourceforge.net/2011/05/building-pocketsphinx-on-android/`

SCORE_DECAY_RATE: If the score of a lattice sentence is lower than one or lower than the the CONF_SCORE_CAP, the SCORE_DECAY_RATE multiplied with the lattice sentence's position is subtracted from the score. The score is then added to the score list.

The strength of the BSF is that it can fix the ASR results if OOV words are present. Its weakness is when the ASR result is wrong, but still matches a bus stop name in step 1. Then, the BSF will return this sentence, as it is a legal result. The BSF can however still manage to get the correct result in later tries, if the returned sentence is blacklisted by the user (see section 6.4).

There is a system parameter called *strong context*, which can be set to true or false by the individual clients using the system. When the *strong context* parameter is set to true, the BSF skips the first step, and does not check whether the first lattice sentence matches a bus stop name. This forces the system to enter step two. By doing so the CBR module is used more often, as the BSF module triggers the CBR module. This can be beneficial if the user has a good case base present.

6.4 The CBR Module

This section explains how the CBR module in TaleTUC works. The CBR module is used to give a solution to a basic problem: given the location and time for a user, where would he or she probably want to go? The problem consists of time and place, and the solution is the destination. When a new case is presented to the CBR module, the search to find the solution for this case starts. This is done by finding the n-closest existing cases from the case database. Before explaining how the n-closest cases are found, some technical knowledge about the system is presented. There are a total of four places where the cases are stored temporarily or long term in the system. What is called the *Case Database* and *Confirmed Cases Database*, are the two long term storage entities. These two databases are maintained as SQL tables, where each entry in a table is a case. Each entry contains the fields: *time-of-day*, *weekend*, *latitude*, *longitude*, *destination* and *device ID*. The fields *time-of-day* and *weekend* constitute the time part of the case, while the *latitude* and *longitude* are the location part of the case. Finally, the *destination* field is the solution for the case, and the *device ID* is used to distinguish users. The two others places where cases are stored are called: *List of Recently Suggested Solutions* and *Blacklisted Solutions*. These two reside in the system's physical memory as arrays of objects at runtime.

The process of finding the n-closest existing cases to a new case is done through an SQL query. As mentioned in the challenges section 3.3, it does not

suffice to simply use the Euclidean distance formula on time-of-day, weekend, latitude and longitude, due to the properties of latitude and longitude. The SQL query was therefore constructed in the way that it calculates the distance between two latitude and longitude points, and uses this distance in an Euclidean distance measurement.

Measuring the distance between two latitude and longitude points is not straightforward because the earth is not flat, it has an oblate spheroid shape. An oblate spheroid shape can be illustrated by taking a soft ball and gently squeezing it between your hands.

It is fairly easy to do calculations on a perfect sphere, as there exist simple and computation friendly formulas for this. However, to perform calculations on an oblate spheroid is more complex. Listing 6.5 shows an example of how Google has implemented this calculation in their Android Location Application Programming Interface (API)[11]. The listing is included directly in the text to give the correct impression on how substantial the calculation need is, the bearing part of the method is however left out, as it is not relevant.

```
computeDistanceAndBearing(double lat1, double lon1,
          double lat2, double lon2, float[] results) {

          int MAXITERS = 20;
          // Convert lat/long to radians
          lat1 *= Math.PI / 180.0;
          lat2 *= Math.PI / 180.0;
          lon1 *= Math.PI / 180.0;
          lon2 *= Math.PI / 180.0;

          double a = 6378137.0; // WGS84 major axis
          double b = 6356752.3142; // WGS84 semi-major axis
          double f = (a - b) / a;
          double aSqMinusBSqOverBSq = (a * a - b * b) / (b * b);

          double L = lon2 - lon1;
          double A = 0.0;
          double U1 = Math.atan((1.0 - f) * Math.tan(lat1));
          double U2 = Math.atan((1.0 - f) * Math.tan(lat2));

          double cosU1 = Math.cos(U1);
          double cosU2 = Math.cos(U2);
          double sinU1 = Math.sin(U1);
          double sinU2 = Math.sin(U2);
          double cosU1cosU2 = cosU1 * cosU2;
          double sinU1sinU2 = sinU1 * sinU2;
```

[11] http://developer.android.com/reference/android/location/package-summary.html

```
        double sigma = 0.0;
        double deltaSigma = 0.0;
        double cosSqAlpha = 0.0;
        double cos2SM = 0.0;
        double cosSigma = 0.0;
        double sinSigma = 0.0;
        double cosLambda = 0.0;
        double sinLambda = 0.0;

        double lambda = L; // initial guess
        for (int iter = 0; iter < MAXITERS; iter++) {
            double lambdaOrig = lambda;
            cosLambda = Math.cos(lambda);
            sinLambda = Math.sin(lambda);
            double t1 = cosU2 * sinLambda;
            double t2 = cosU1 * sinU2 - sinU1 * cosU2 *
               cosLambda;
            double sinSqSigma = t1 * t1 + t2 * t2; // (14)
            sinSigma = Math.sqrt(sinSqSigma);
            cosSigma = sinU1sinU2 + cosU1cosU2 * cosLambda; //
               (15)
            sigma = Math.atan2(sinSigma, cosSigma); // (16)
            double sinAlpha = (sinSigma == 0) ? 0.0 :
                cosU1cosU2 * sinLambda / sinSigma; // (17)
            cosSqAlpha = 1.0 - sinAlpha * sinAlpha;
            cos2SM = (cosSqAlpha == 0) ? 0.0 :
                cosSigma - 2.0 * sinU1sinU2 / cosSqAlpha; //
                   (18)

            double uSquared = cosSqAlpha * aSqMinusBSqOverBSq;
               // defn
            A = 1 + (uSquared / 16384.0) * // (3)
                (4096.0 + uSquared *
                 (-768 + uSquared * (320.0 - 175.0 * uSquared)))
                    ;
            double B = (uSquared / 1024.0) * // (4)
                (256.0 + uSquared *
                 (-128.0 + uSquared * (74.0 - 47.0 * uSquared)))
                    ;
            double C = (f / 16.0) *
                cosSqAlpha *
                (4.0 + f * (4.0 - 3.0 * cosSqAlpha)); // (10)
            double cos2SMSq = cos2SM * cos2SM;
            deltaSigma = B * sinSigma * // (6)
                (cos2SM + (B / 4.0) *
                 (cosSigma * (-1.0 + 2.0 * cos2SMSq) -
                  (B / 6.0) * cos2SM *
                  (-3.0 + 4.0 * sinSigma * sinSigma) *
                  (-3.0 + 4.0 * cos2SMSq)));
```

```
            lambda = L +
                (1.0 - C) * f * sinAlpha *
                (sigma + C * sinSigma *
                 (cos2SM + C * cosSigma *
                  (-1.0 + 2.0 * cos2SM * cos2SM))); // (11)

            double delta = (lambda - lambdaOrig) / lambda;
            if (Math.abs(delta) < 1.0e-12) {
                break;
            }
        }

        float distance = (float) (b * A * (sigma - deltaSigma));
    }
```

Listing 6.5: Google's calculation on oblate spheroid shapes

The method in listing 6.5 is based on formulas proposed by Vincentry (1975). It involves several iterations of calculations to reduce the calculation error. This method is too computation heavy to be used to calculate the distance in TaleTUC's CBR module. The reason for this is that the method would have to be run for each relevant case in the case base, where relevant cases are cases that match the given device ID. Luckily, the earth's oblate spheroid shape is not very significant. According to the WGS 84 World Geodic System standard (NIMA, 2000), the radius at the equator is 6378137 metres, and the radius at the poles is 6356752 metres. This results in a difference of about 22 kilometres. Because the difference is so small, this shape can be ignored in the calculation of small distances. The distance calculation on spheres is often called the great-circle distance[12] calculation. There are a number of formulas to choose from when calculating the great-circle distance, where some have rounding errors for small distances, some are computational heavy and some have rounding errors when dealing with antipodal points[13]. The one chosen for TaleTUC is called the *haversine* formula. The *haversine* is computation friendly, and its only drawback is that it has rounding errors when dealing with antipodal points (Sinnott, 1984). Calculation problems with antipodal points can be ignored as the geographical field of operations for TaleTUC is very small compared to the size of the earth. The *haversine* formula looks like this:

$$haversin(\frac{d}{R}) = Haversine(\Delta\phi) + cos(\phi_1)cos(\phi_2)haversin(\Delta\lambda) \qquad (6.1)$$

Where R is the radius of the sphere, ϕ_1 and ϕ_2 is latitude points, $\Delta\phi$ is ϕ_1 - ϕ_2,

[12] http://mathworld.wolfram.com/GreatCircle.html
[13] http://www.antipodemap.com/

$\Delta\lambda$ is the difference between two longitude points and d is the distance between the two points (along the great circle arc).
The *haversine* function looks like this:

$$haversin(\theta) = \frac{1 - cos(\theta)}{2} \tag{6.2}$$

Listing 6.6 shows the *haversine* formula 6.1 solved for d, and translated into an SQL query.

```
SELECT *, 6378137 * 2 * ASIN(SQRT( POWER(SIN((orig_lat -abs(dest.
   lat)) * pi()/180 / 2),2) + COS(orig_lat * pi()/180 ) * COS(abs
(dest.lat) * pi()/180) * POWER(SIN((orig_lon - dest.lon) *  pi()
   /180 / 2), 2) )) as distance;
```

Listing 6.6: The haversine formula solved for d, and translated into an SQL query

Listing 6.7 shows the SQL query when the rest of the variables used in TaleTUC are included:

```
SELECT *, 6378137 * 2 * ASIN(SQRT( POWER(SIN((orig_lat -abs(dest.
   lat)) * pi()/180 / 2),2) + COS(orig_lat * pi()/180 ) * COS(abs
(dest.lat) * pi()/180) * POWER(SIN((orig_lon - dest.lon) *  pi()
   /180 / 2), 2) )) as distance FROM QUERY where devID = 'tester2'
     ORDER BY SQRT(POW(distance,2)+POW((QUERY.timeOfDay -
   currentTimeOfDay),2) +POW((QUERY.weekend - currentWeekend),2))
   LIMIT 10;
```

Listing 6.7: The haversine SQL query with variables

When the n-closest cases are found, the construction of the solution for the new case is started. The n-closest cases are ordered so that the first item in the list is the closest match. The cases are scored on their position in the list. This scoring is dependant on the system variables called *N_ CASES_ TO_ CONSIDER* and *SCORE_DECLINE_RATE*. Lets say that the *N_CASES_TO_ CONSIDER* is set to 10 and the *SCORE_DECLINE_RATE* is set to 1. The first item in the list gets 10 as its score, the next item in the list gets 9 and so on. If case number one and three in the list have the same case solution, it means that this solution now has a total score of 18. This feature makes sure that the solution to the new case is a generalisation of the n-closest case solutions.

In relation to system variables, there are three additional ones. The two first are called *TIME_WEIGHT* and *GEO_WEIGHT*. These two make it possible to weight either time or geographical location in the case matching. The last system variable is called *GEO_FIELD_OF_OPERATION*. It defines the max diameter of the system's field of operation in metres. This variable is used by the CBR

module to adjust the matching values, so that time and location are equally weighted (if not altered by *TIME_WEIGHT* or *GEO_WEIGHT*).

Before the new case is finalised, scores from the *Feedback Case Database* are added to the total solution score. The scores from the *Feedback Case Database* are fetched in a similar way to how the *n*-closest cases are fetched from the *Case Database*. In this way, the knowledge from *revised* cases is added to the final case solution.

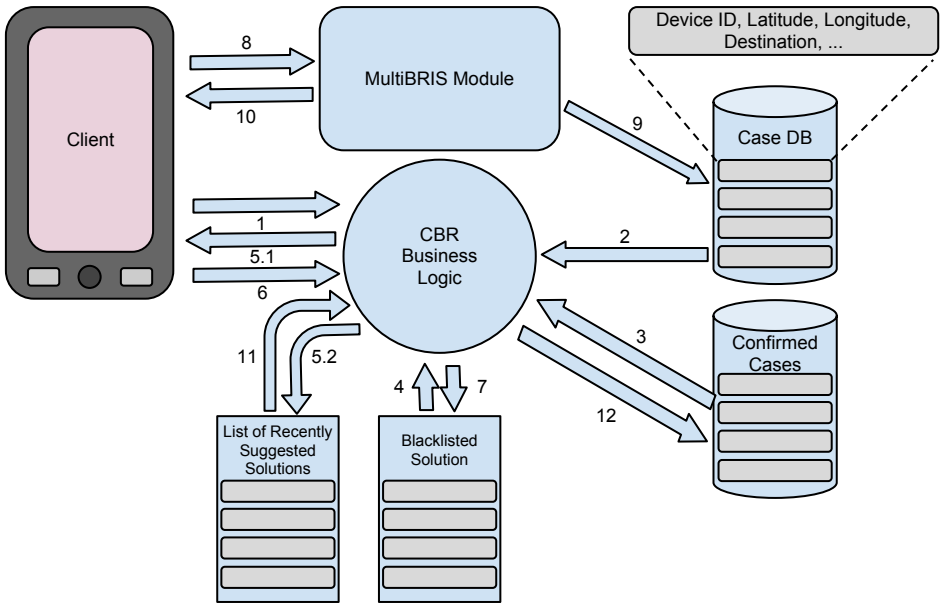

Figure 6.3: CBR Module

The following explains what happens at each step in figure 6.3.

1. The client sends its device ID, latitude and longitude to the CBR-system on the pretence to get a suggested destination in return.

2. The CBR business logic fetches the most promising cases from the *Case Database*. This is a part of the *retrieve* process discussed in section 3. The *n*-closest matched are found as described earlier.

3. The CBR business logic fetches the most promising cases from the *Confirmed Cases Database*. This is done similarly to how the cases from the *Case Database* are fetched. If a solution in the list from the *Case Database* also exists in the *Confirmed Cases Database*, that solution is credited with more score, dependant on the solution's position in the list from the *Confirmed Cases Database* list. This way, the system builds it's new solution for the

new problem case. This is part of the *reuse* process. The system builds itself a ranked list of this possible case solution, so that if the best found solution is for instance blacklisted, the next best case solution can be used.

4. The blacklist is checked if the best solution found is tagged as blacklisted for the current device ID. If the best solution turns out to be blacklisted, the next case solution is selected to be sent back to the client.

5. The best case solution is sent back to the client. The best solution is also temporarily stored in the list called: *List of Recently Suggested Solutions*. This list is used in the *revise* process of the CBR system. A case solution exists in the *List of Recently Suggested Solutions* for a time span dependant on a system parameter called *FEEDBACK_ SUSTAIN_ INTERVAL*.

6. If it turns out that the case solution was wrong, the client reports this back to the system.

7. A case solution is reported to be wrong and unwanted for a particular device ID. The solution is therefore temporarily stored in the blacklist, so that the same error cannot occur again within a short period of time.

8. If the case solution received by the client is correct, the normal usage pattern is then to use this solution, or more specific, the destination to do a bus route query to the MultiBRIS module.

9. The MultiBRIS module stores the MultiBRIS bus route query as a case in the *Case Database*.

10. The MultiBRIS module replies bus route information to the client.

11. The *revise* and *retain* parts of the CBR system check the *List of Recently Suggested Solutions* against the *Case Database* consecutively. The system checks if the device ID and the solution from any new case matches a device ID and a solution in the *List of Recently Suggested Solutions*. If it finds a match, it means that this solution was given to the specific device ID, not long ago. It also means that a suggested case solution was confirmed by the client, and it is therefore successfully revised and found correct.

12. If a case solution is found revised and correct, it is stored in the *Confirmed Cases Database*. The case solution is also removed from the *List of Recently Suggested Solutions*, even if the time span of the *FEEDBACK_ SUSTAIN_ INTERVAL* is not reached.

6.5 The Combined Modules

The connection link between the ASR and CBR modules is the BSF module. As described in section 6.3, the BSF creates an ordered list of scored answers. Before the final answer is returned to the user, the system checks the scores and the confidence scores of the bus stop names in this list. If the first two answers in the list have the same scores, and the same confidence scores, the system deducts that the probability that the right answer is found is too low. The system then uses the CBR module to get an answer instead, which is returned to the user.

6.6 TABuss as a TaleTUC client

In the use with TaleTUC, TABuss represents the client component in figure 6.2. Two audio processing functionalities were created, in addition to graphic components, in TABuss to use it as a TaleTUC client. The first was a recording method. This method records audio, and stores a wav file on the SD-card. The second extracts the cepstrum from this wav file, and stores this in a separate file. Which one of these files to send to TaleTUC is controlled in the source code. The method that extracts the cepstrum uses the *Sphinx-4* front end configuration file in listing 6.3. This file is downloaded to the device's SD-card on start-up, if it does not already exist.

As seen in figure 6.2, a query to TaleTUC consists of more than only an audio file. In addition to the device ID, location information is needed. This is retrieved from the device's location technology (GPS, WiFi, etc). The process of running the ASR, and using the result in TABuss' query functionality is depicted in figure 6.4.

User verification was added as a measure to limit the user frustration when the system returns the wrong results. The user can verify the result through a touch event in the answer screen. If the user accepts the ASR result, a query is sent to the MultiBRIS server, with the result as one of the inputs. If the user does not accept the ASR result, the recording process is restarted automatically. At the same time, the ASR result is blacklisted (see section 6.4). This has a limit of two tries. A third try will display a message stating that the system cannot provide the correct ASR result, and that the user should type in the correct bus stop name.

In addition to the use of TABuss as a TaleTUC client, it was desirable to further explore the smartphone capabilities. As a result, a widget[14] was developed. This home screen widget is a part of TABuss, but can provide function-

[14] `http://developer.android.com/guide/topics/appwidgets/index.html`

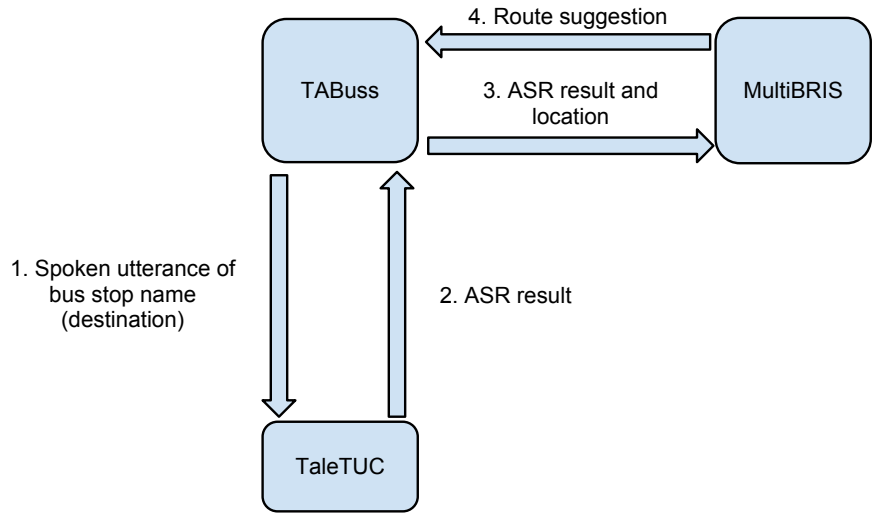

Figure 6.4: TABuss' query based on ASR results

alities without the need to start the application. At this moment, the widget uses TABuss for the calculation of route suggestions through MultiBRIS, and the display of these. The widget's main functionality is depicted in figure 6.5. The numbers indicate the order of the performed operations.

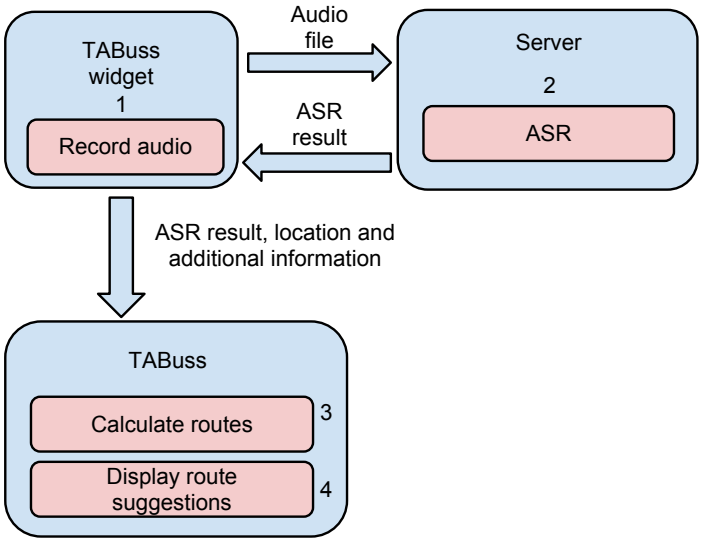

Figure 6.5: The TABuss widget architecture

Chapter 7

Results

This chapter describes TaleTUC's development results.

7.1 The ASR and the BSF Modules

The following sections first describe the developed *language models* and *acoustic models*. The results from the prototype comparisons of *Sphinx-4* and *PocketSphinx* are then presented. Finally, the development results of the *Sphinx-4* solution, and results from the BSF module are described.

7.1.1 ASR Models

The 50 bus stop names in the *language model* are shown in listing 7.1. These were in the text corpus which was used as input to the *language model* training program.

```
alfheimsvingen
anders buens gate
anton jenssens veg
brøsetvegen
brekkåsen
brobakk
brukseier olsens vei
bukkvollan
dronningens gate
eggan klæbu
einar tambarskjelves gate
fannrem stasjon
gafset
gimseflata
gløshaugen
```

```
granåsen gård
hallfred høyems veg
ila
ilsvika
kambru
kroppanmarka snuplass
kvitland melhus
langlo
lundåsen
målsjøåsen
marcus thranes vei
nova kinosenter
nypantunet
omkjøringsveien nardo
osveien
persaunevegen
pirbadet
prestgårdskrysset
presthusaunet
rønningen aldersbolig
ranheim idrettsplass
rydningen
sjøla
skårgangen
sluppen
sollia
svebergkrysset
tanemskrysset
trøndertun
trondheim sentralstasjon
udduvoll bru øst
udduvoll bru vest
voll studentby
øie skole
øysandkrysset
```

Listing 7.1: The bus stop names in the language model

The outputted *language model* file was after completion added to the path of the *acoustic model* building program. This program was then run to find the best parameter values for the number of *senones*, and the final number of *densities*. The result from these runs were within *Sphinx*' recommendations, given the size of the training data. Higher numbers would make the model over-trained, which means the system would be poor at recognising unseen speech (i.e., speakers it is not trained for). The following settings were therefore chosen:

Number of *senones*: 25

Final number of *densities*: 6

Screenshots of the Android application developed for recording the training data are shown in figure 7.1, where the recording process is illustrated. The numbers indicate the order of the performed operations.

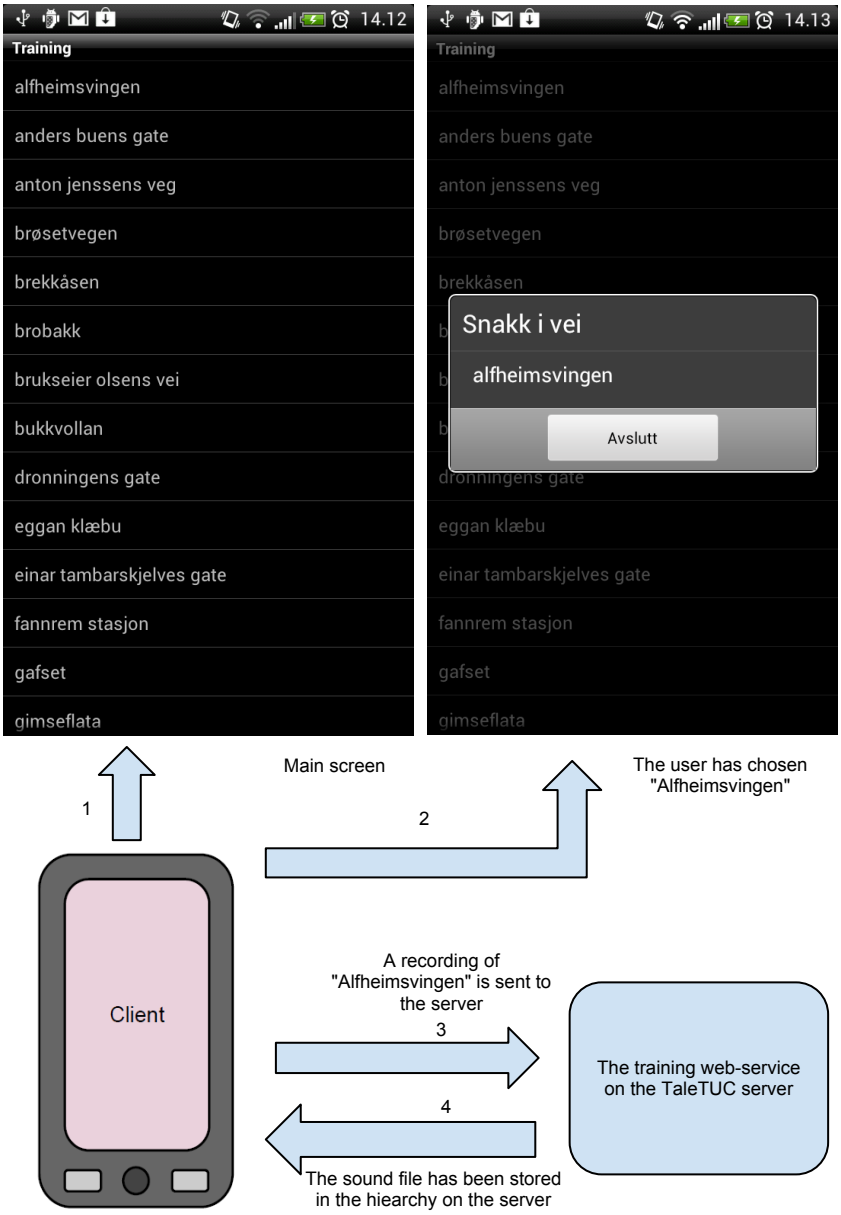

Figure 7.1: The recording application that collects data for the *acoustic model*

7.1.2 Prototype Comparisons of Sphinx-4 and PocketSphinx

The results from the comparison between the *Sphinx-4* solution and the *PocketSphinx* solution showed the same recognition results. This was because both of the solutions used the same *acoustic models* and *language models*, and because the front-end used to compute the cepstrums performs the same operations as *Sphinx-4* does in its built-in cepstrum computations. The advantage of the *PocketSphinx* implementation was that it returned results quicker, which was expected as no sending of sound files to a server was necessary. The disadvantage was the need for free storage space on the test device's SD-card. 1,2 MB of storage space was required for a vocabulary of 50 words, and a *language model* trained with four speakers. A larger vocabulary and a more robust *acoustic model* would require a larger amount of free storage space available.

Results in the *Sphinx-4* program were returned in an average of three seconds, when the application was tested in a WiFi network. The testing was performed before the extraction of cepstrums was implemented in the *Sphinx-4* front-end configuration, and was therefore done with wav files. The tested wav files had an average size of 75 kB. In addition, when HTTP headers are added, the size of each file transfer increases. But compared to the *PocketSphinx* solution, the storage of *language models* and *acoustic models* was avoided.

The size of the needed available storage space in the *PocketSphinx* prototype lead to a reinforced impression of *Sphinx-4* and a client-server archictecture being the best choices for TaleTUC. Also, the use of the Android NDK increased the application complexity, as it introduced an additional dependency. Developers need to use two different programming languages (Java and C) to create *PocketSphinx* programs. This requires either background knowledge of the additional programming language, or that there is time available to learn it.

7.1.3 ASR Implementation with Sphinx-4

Both of the trained *acoustic models* and three test sets were used in the performed runs. ”Clean” means that the *acoustic model* was trained indoors, with minimum background noise. ”Mixed” means an *acoustic model* where the ”clean” training sets have been mixed with training sets that contain audio recorded outdoors. For the test sets, ”indoors” means that the audio was recorded indoors, and ”outdoors” means that the audio was recorded outdoors. ”Mixed” means a test set where every other audio file is recorded indoors and outdoors.

For the BSF the parameters in Table 7.1 were chosen:

Parameter name	Parameter value
CONF_SCORE_CAP	1
SCORE_DECAY_RATE	2

Table 7.1: The BSF parameters used for the WER tests

The runs presented in Table 7.2 and Table 7.3 were done without any form of noise cancellation or filters.

Trained model	WER indoors	WER outdoors	WER mixed
Clean	10 %	66 %	38 %
Mixed	10 %	32 %	18 %

Table 7.2: ASR without filters

Trained model	WER indoors	WER outdoors	WER mixed
Clean	0 %	52 %	26 %
Mixed	4 %	20 %	12 %

Table 7.3: ASR without filters, optimised with the BSF

Based on these tables, the best results were with the mixed *acoustic model*. The remaining tests therefore focused on this model. The results also showed that the BSF provided optimised WERs for all of the tests, and even achieved 0 % WER in one of them. The current tests were conducted with both the cepstrum decoding front-end and the wav-file decoding front-end. Because of the reduced data traffic achieved with the cepstrum front-end, shown in section 7.4.2, the remaining tests and results in this section are based on this front-end.

Next, *Sphinx-4*'s *preemphasizer* and *speechClassifier* were introduced in the project's configuration file. For finding the optimal threshold in the *speechClassifier* property, several tests were performed. As seen from figures 7.2, 7.3 and 7.4, a low threshold gave the best results. A value of two was therefore chosen, which achieved the lowest WERs of all the tested thresholds. It was assumed that thresholds higher than 30 would not give lower WERs, as the use of thresholds up to and including 30 showed a trend with an increasing WER.

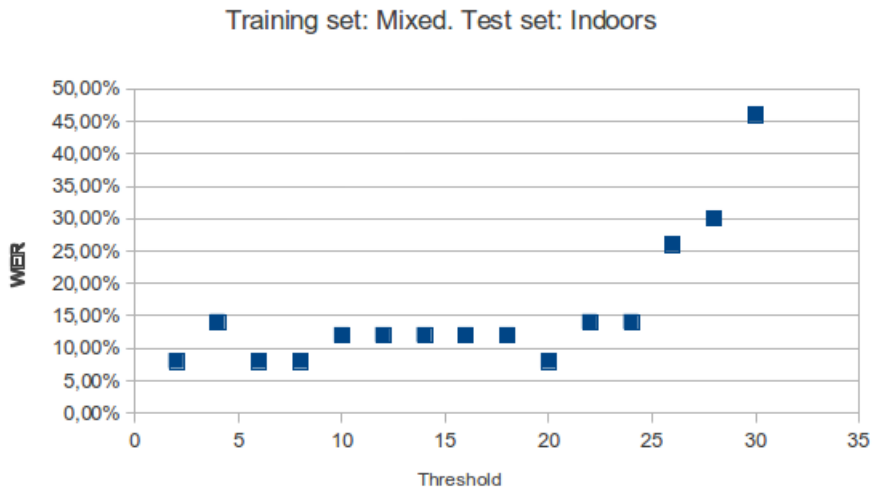

Figure 7.2: VAD threshold test 1

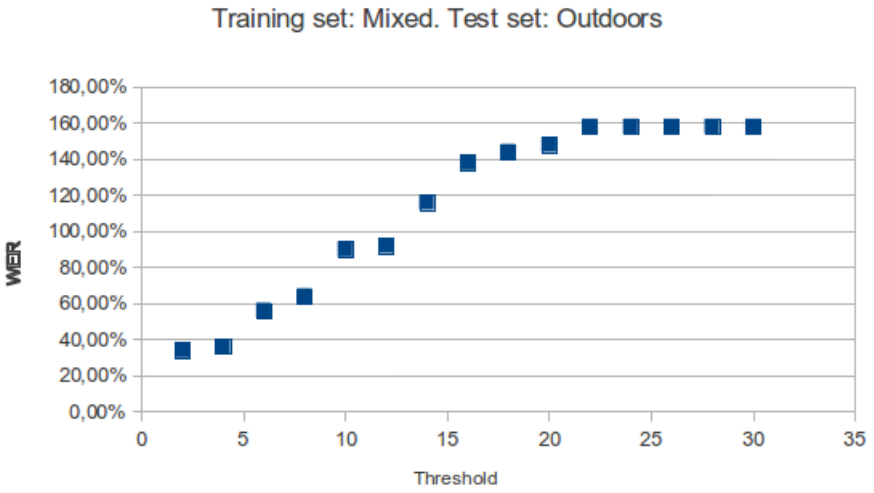

Figure 7.3: VAD threshold test 2

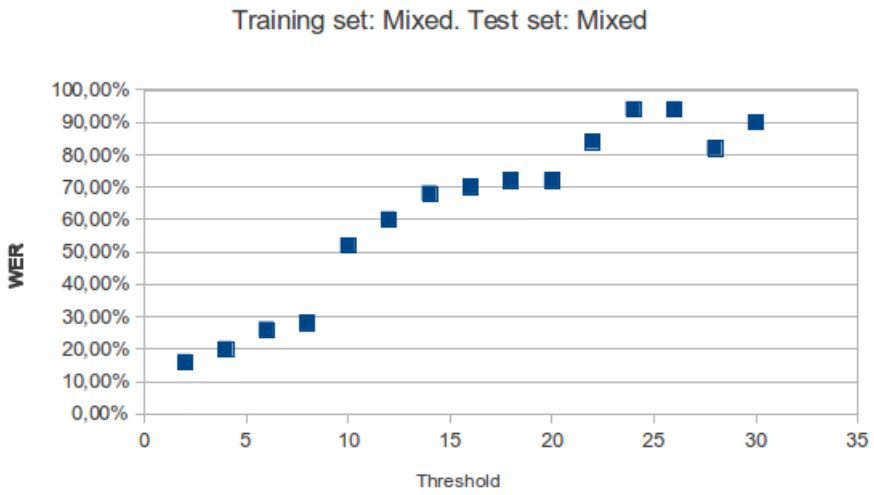

Figure 7.4: VAD threshold test 3

Next, a *Wiener filter* was implemented. The new *acoustic model* was trained on *Wiener filtered* audio, where the *speechClassifier* threshold was set to two, which was the best result from the threshold tests. The system was then tested, where the *speechClassifier* property for the test program had to be tuned, to achieve the lowest possible WER with the new *acoustic model*. The threshold found to give the best results for all of the three test sets was five. Table 7.4 displays the WER of the performed runs with a *Wiener filter*, and Table 7.5 displays the same runs optimised with the BSF.

Trained model	**WER indoors**	**WER outdoors**	**WER mixed**
Mixed	6 %	26 %	10 %

Table 7.4: ASR with a Wiener filter

Trained model	**WER indoors**	**WER outdoors**	**WER mixed**
Mixed	0 %	22 %	4 %

Table 7.5: ASR with a Wiener filter, optimised with the BSF

Table 7.6 displays the comparisons of the WERs achieved with and without a *Wiener filter*. As can be seen, the runs with a *Wiener filter* achieve lower WERs for all of the tests. Table 7.7 displays the same comparisons, where the BSF has been used. As one can see, a lower WER is achieved with the use of a *Wiener filter*, compared to without, in all tests, except for one.

Trained model	Test set	WER without filters	WER with a Wiener filter
Mixed	Indoors	10 %	6 %
Mixed	Outdoors	32 %	26 %
Mixed	Mixed	18 %	10 %

Table 7.6: Comparison of the WERs without filters, and the WERs with a Wiener filter

Trained model	Test set	WER without filters	WER with a Wiener filter
Mixed	Indoors	4 %	0 %
Mixed	Outdoors	20 %	22 %
Mixed	Mixed	12 %	4%

Table 7.7: Comparison of the WERs without filters, and the WERs with a Wiener filter, optimised with the BSF

7.2 The CBR Module

This section contains two different test results: one is based on a very practical test where a natural usage pattern was set up, while the other was a more scientific approach using *n*-fold cross-validation.

The first test scenario, named "the natural usage scenario", for the CBR module was as follows: the user is located at three different places during one day. Depending on the time-of-day and the user's location, the user wants to go to one of three places. The two locations the user is located at are approximately 2626 metres apart. The user is located at "Ila" at 09:00 and wants to go to "Gløshaugen". The user is located at "Gløshaugen" at 14:00 and wants to go to "Dragvoll". At 16:00 the user is located at "Gløshaugen" and wants to go to "Ila". The trip from "Dragvoll" back to "GlÃ¸shaugen" is omitted. This scenario setup is illustrated in figure 7.5. The case base for testing was filled by using this scenario as a starting point. Deviations from a uniform case base were then created by adding random variations in time and geographical locations. To represent the case where a user, for one or more days, decided to deviate from his or her usual travel pattern, a "noise" percentage function was added. The noise percentage function works by giving each case in the case base a chance at being replaced with a wrong solution. The chance for this to be replaced depends on one of the input parameters used in the test. This input parameter was simply called "noise", and was added to simulate an abnormal daily schedule by a user. There was a total of three input parameters that were iterated through to cre-

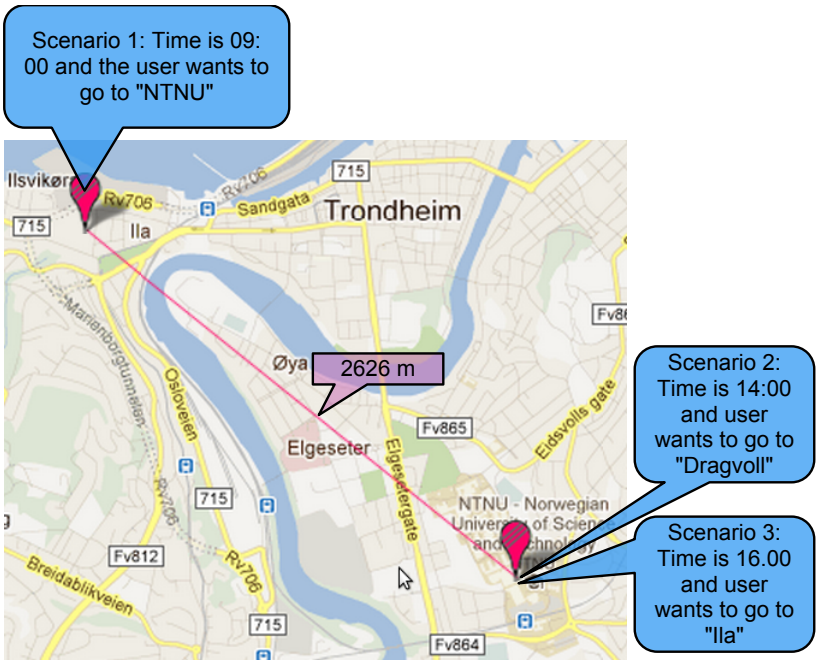

Figure 7.5: Graphical representation of the test setup

ate the results in this section, namely time, location and noise variations. The parameters in figure 7.9 are all converted into percentages to make both input parameters and the results displayable in the same graph. The results were produced by checking if the right destinations were suggested by the system, for base scenarios.

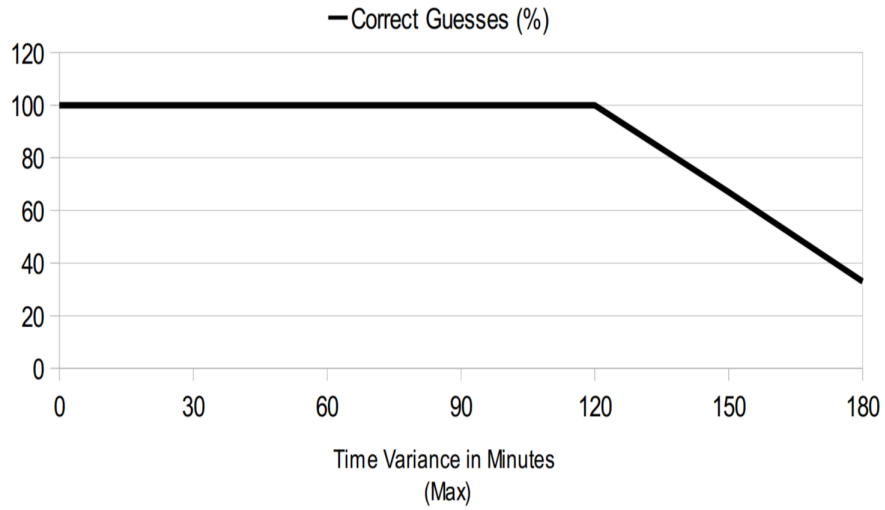

Figure 7.6: Results with different time variations in case set, when using the "Natural Scenario" setup

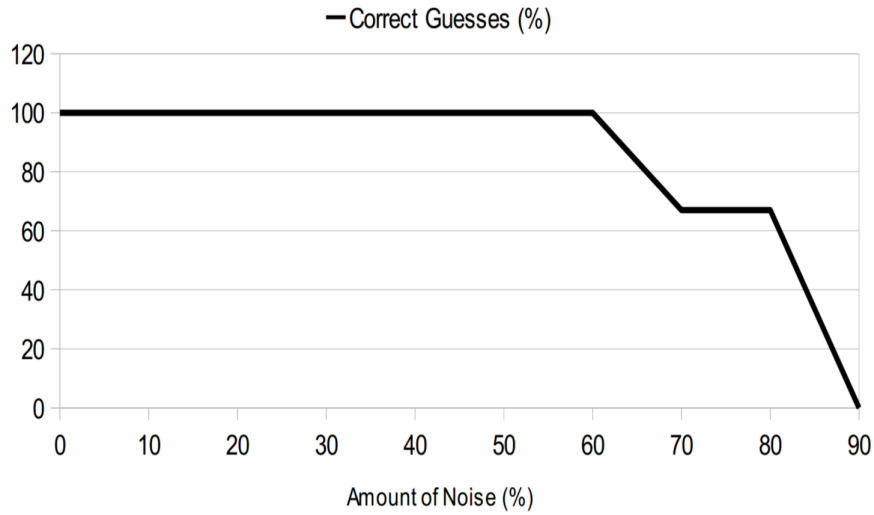

Figure 7.7: Results with different noise percentages, when using the "Natural Scenario" setup

The next test is called a *n*-fold cross-validation test, had a similar base setup, where the base scenario consisted of 6 core cases. These scenarios were set up so that they tested the two key concepts for the CBR system: travelling to different destinations from the same location, only on different times, and travelling to a different location on the same time, but from different locations. 200 different cases were generated with different properties dependant on the re-

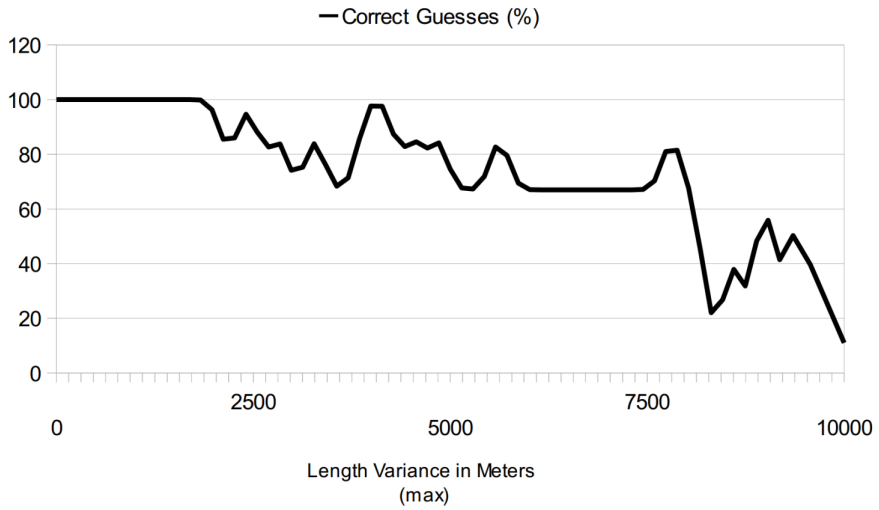

Figure 7.8: Results with different distance variations, when using the "Natural Scenario" setup

spective tests. These created the basis for the 10-fold cross validation. 10-fold cross validation uses 90% of the available data to train the case base, and 10 % to test it with, which is said to be optimal when using *n*-fold cross validation. Kohavi (1995) compared several approaches to estimate accuracy in his paper, *A study of cross-validation and bootstrap for accuracy estimation and model selection*. He tested: cross-validation (including regular cross-validation, leave-one-out cross-validation , stratified cross-validation) and bootstrap (sample with replacement), and recommended stratified 10-fold cross-validation as the best method, as it tends to provide less biased estimations of the accuracy. The tests for TaleTUC's CBR module were performed with 2x10-fold cross validation, and are shown in figures 7.10, 7.11 and 7.12. The "2x" means that each test was run twice per new parameter setting. As the entire dataset was replaced between the two tests, it gave an even more correct system accuracy. The system settings used for the test can bee seen in Table 7.8.

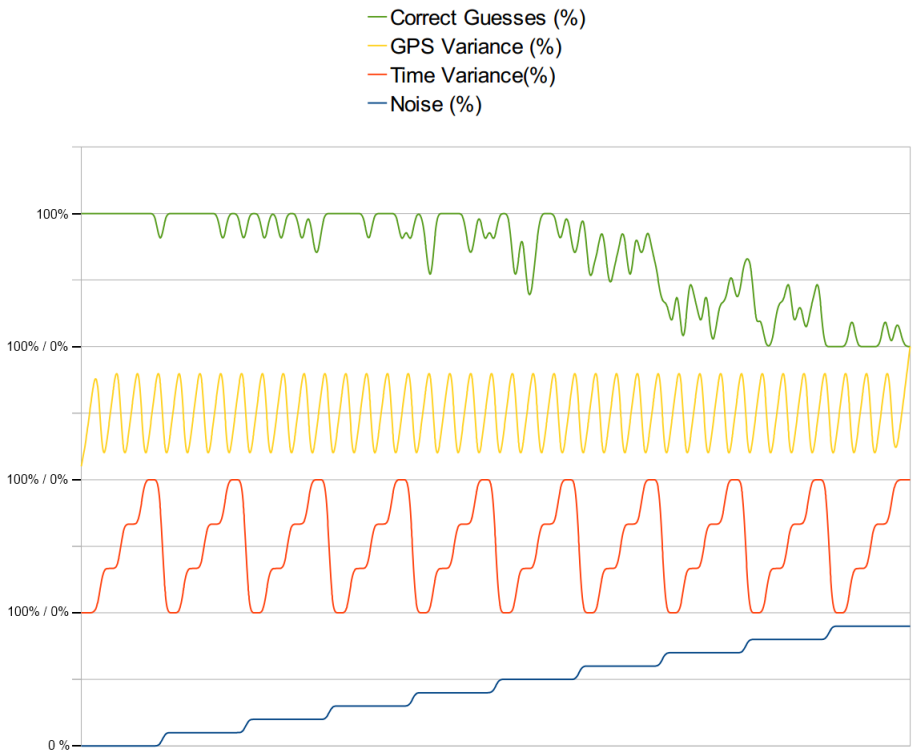

Figure 7.9: CBR Result where time, noise and distance variations are combined. 100% GPS Variance: 0.02 (3143 metres), 100% Time Variance: 45 minutes, using the "Natural Scenario" setup

System setting name	**Setting value**
GEO_ FIELD_ OF_ OPERATION	15000
TIME_ WEIGHT	1
GEO_ WEIGHT	1
N_ CASES_ TO_CONSIDER	10
SCORE_ DECLINE_ RATE	1

Table 7.8: System settings for the CBR module test

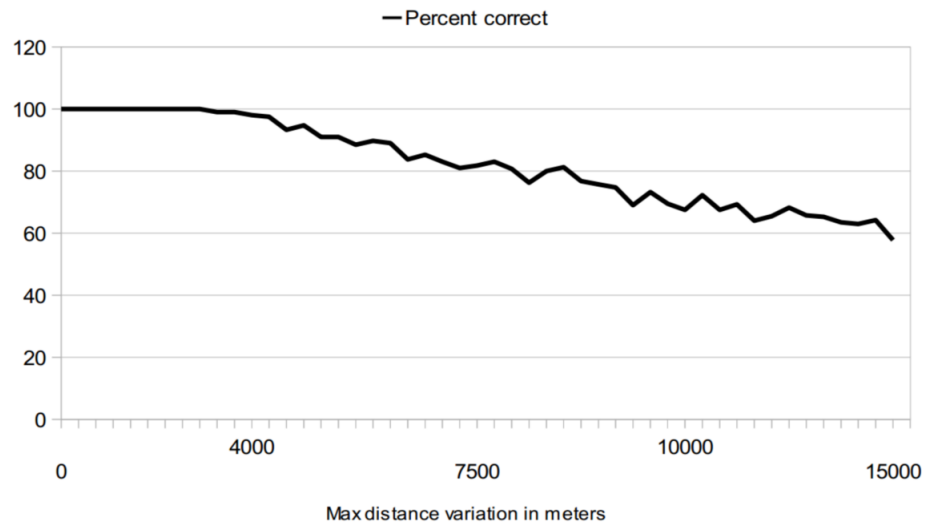

Figure 7.10: Correct guesses with distance variations, when using 2x10-fold cross validation. Case base includes 200 cases

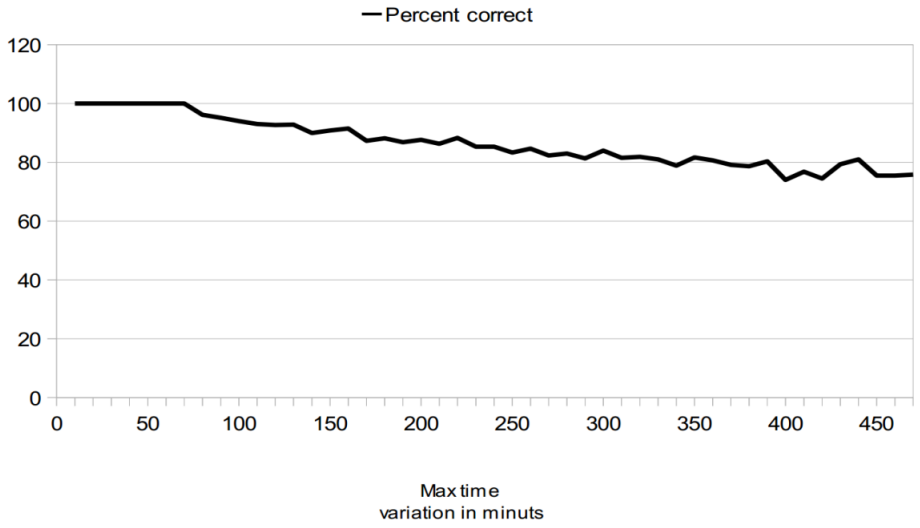

Figure 7.11: Correct guesses with time variations, when using 2x10-fold cross validation. Case base includes 200 cases

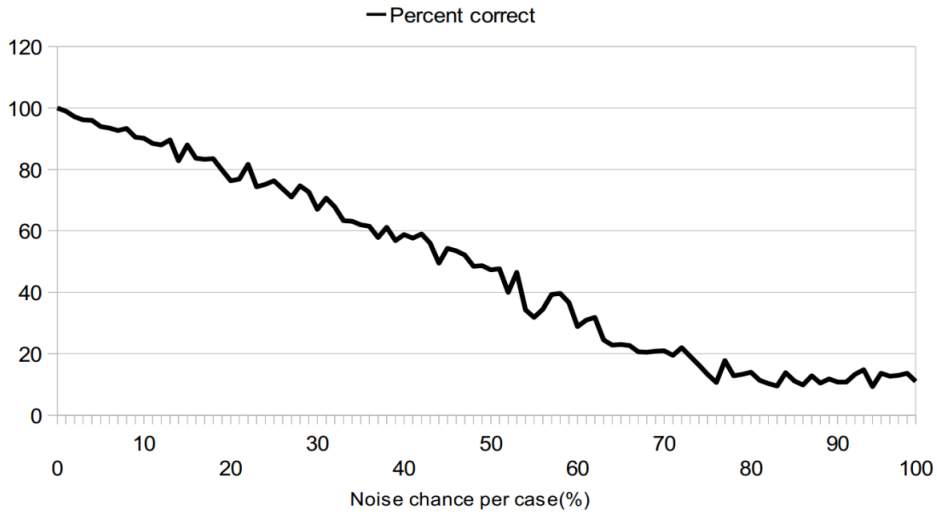

Figure 7.12: Correct guesses with noise amount, when using 2x10-fold cross validation. Case base includes 200 cases

7.3 The Combined Modules

This section contains the test results from all three of the systems modules combined. In order to test the ASR, BSF and CBR modules together, a CBR case base was trained to fit the available audio test data. As this audio test data was used to test the ASR and BSF, it provided a base for testing if the CBR module could improve the WERs. The audio test data consisted of 100 sound samples. 50 were recorded indoors, and 50 were recorded outdoors, as explained in the ASR result section (section 7.1.3). A "core case" was then generated for each of these audio samples. What is referred to as a "core case", is a point in a user's daily bus travel schedule. Because it is unnatural to have 100 "core cases" for one user (a user is distinguished by the device ID as explained earlier), the 100 cases were devised into 17 sections (where each section can be viewed as a "user's" set of "core schedules"). By doing so, each section except the last one contained six "core cases". The six "core cases" were designed to test both time and place overlapping cases, as explained in the CBR method section (section 6.4). For each "core case", ten cases were generated on the basis of the settings in Table 7.9. This gave approximately 60 cases for each case set, and a case base with 1000 cases in total. A case set is a set of cases that belongs to one user.

The test setup used the optimal ASR settings as described in section 7.1.3. Table 7.10 presents the WER results from four different combinations of the system modules. The two last columns represent the WERs when all three of the system modules were used together (the ASR module, the BSF module and the CBR module). The first two columns show the WER results from the ASR and the BSF modules. The first two columns are included to give an easy comparison of the WER results for all the system modules. As can be seen from Table 7.10, combining the CBR module with the ASR and the BSF modules gave a 2% point reduction in the WER in a mixed sound environment. The largest improvement in WER was produced when the CBR module with the *strong context* setting was applied to the test set recorded outdoors. This yielded an improvement of 18 % points in the WER.

Settings name	**Setting value**
GEO_ FIELD_ OF_ OPERATION (CBR)	15000
TIME_ WEIGHT (CBR)	1
GEO_ WEIGHT (CBR)	1
N_ CASES_ TO_CONSIDER (CBR)	10
SCORE_ DECLINE_ RATE (CBR)	1
SCORE_ DECAY_ RATE (BSF)	2.0
CONF_ SCORE_ CAP (BSF)	1.0
Geographical variance (CBR)	1500 metres
Time variance (CBR)	30 minutes
Noise chance per case (CBR)	20 %
SpeechClassifier threshold for Wiener filtered training set (ASR)	2
SpeechClassifier threshold for test set (ASR)	5
Acoustic model (ASR)	Mixed

Table 7.9: System and test settings for the combined module test

Test set	**ASR WER**	**ASR with BSF WER**	**Combined modules WER**	**Combined modules WER with *strong context***
Indoors	6 %	0 %	0 %	0 %
Outdoors	26 %	22 %	18 %	4 %
Mixed	10 %	4 %	6 %	0 %

Table 7.10: Comparison between the WER results of the ASR module, the ASR module with the BSF and the combined modules

7.4 The TaleTUC Client

The following sections present screenshots and descriptions of the TaleTUC client. The data traffic and performance are then described.

7.4.1 Screenshots and Descriptions

Figure 7.13 displays the recognition process from TABuss' perspective. First, the audio is recorded by pressing a button in the home screen menu. The audio (either a wav file or a cepstrum file) is then sent to the TaleTUC server. Finally, an answer is returned to the user, and displayed in an answer screen. The text in the first screenshot translated to English is: *Speak* (first line), *Press button when done* (second line) and *Done* (Button text).

Figure 7.14 displays the event when the user has pressed the ASR result text field. The user is then prompted with a dialog asking for verification, in this case for "ila". If the user chooses *ja* (yes), TABuss displays route suggestions, as seen in the figure. If the user chooses *nei* (no), TABuss performs the operations described in section 6.6, on verification. If the number of tries exceed the set limit of two tries, the user is asked to manually type the wanted destination.

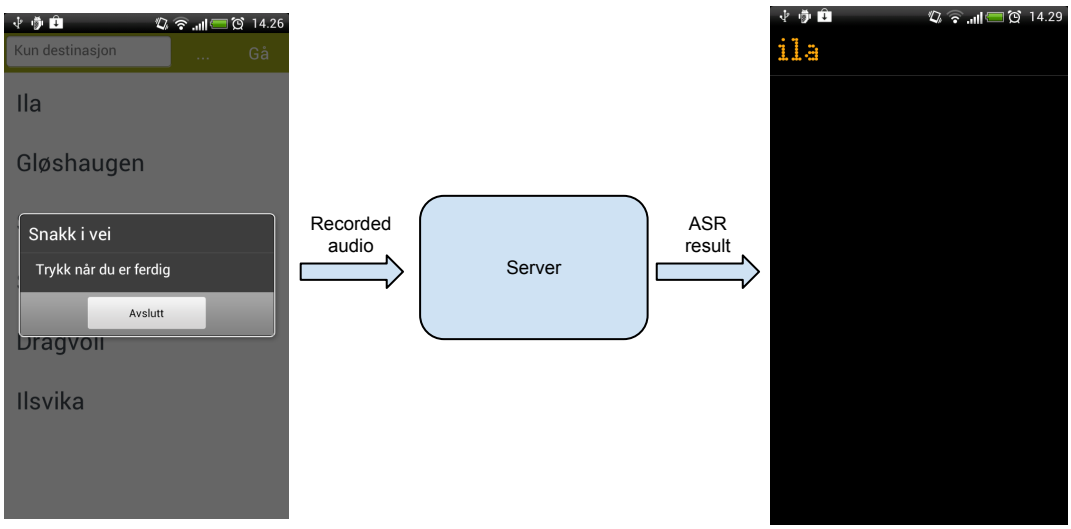

Figure 7.13: TABuss speech input and ASR result

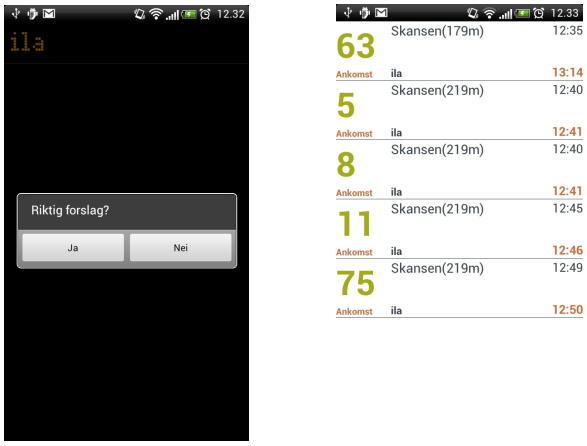

Figure 7.14: TABuss ASR result and calculated route suggestions

7.4.2 Data Traffic and Performance

This section describes the data traffic and performance of the wav decoder (cepstrums are computed on the server), and the cepstrum decoder (where cepstrums are computed on the device). Table 7.11 lists the results from five runs for each of the solutions, with five bus stop names of different lengths. The runs were performed with the test devices, where the audio data was transmitted over a WiFi connection. The data traffic sizes listed do not account for HTTP headers or other additional data required for transmissions.

Phrase	**Wav file size**	**Cepstrum file size**	**Wav time use**	**Cepstrum time use**
Gløshaugen	42,3 kB	6,8 kB	2,58 sec	4 sec
Einar Tambarskjelves gate	84,5 kB	13,7 kB	3,93 sec	6,85 sec
Ila	46,1 kB	7,4 kB	2,35 sec	3,98 sec
Ilsvika	38,4 kB	6,2 kB	2,72 sec	3,48 sec
Trondheim sentralstasjon	65,3 kB	10,6 kB	3,36 sec	4,81 sec

Table 7.11: Performance comparisons of the wav and cepstrum solutions

As seen from Table 7.11, there is a trade-off between the time use and the data traffic. The cepstrum solution performes slower than the wav-solution, but makes up for it by providing approximately six times smaller file sizes for the tested audio files. Also, to illustrate the size difference between the wav files and the cepstrum files: the wav recordings used for training the mixed *acoustic model* had a combined size of 40,1 MB. The extracted cepstrums of these had a combined size of 5,7 MB. This means that there will be a significant difference in the data traffic with regular use.

7.5 The TaleTUC Client Widget

Figure 7.15 displays the TABuss client widget. The image in the middle shows the start screen. The image to the left displays the event that occurs when the icon is pressed, where the user can record audio. The text translated to English is: *Hold to speak* (text above mic). The image to the right displays the event that occurs when the text is pressed, which is to open the TABuss application.

Figure 7.16 displays the event when the user has recorded audio. The audio file (wav or cepstrum) is then sent to the TaleTUC server (as seen in figure 6.5). The same as with the TaleTUC client is then seen, where the user is prompted with a verification dialog. If the user chooses *Riktig* (correct), TABuss displays route suggestions, as seen in the figure. If the user chooses *Feil* (wrong), the widget performs the operations described in section 6.6, on verification. If the number of tries exceed the set limit of two tries, the widget opens the TABuss home screen. From here, the user can manually type in the destination.

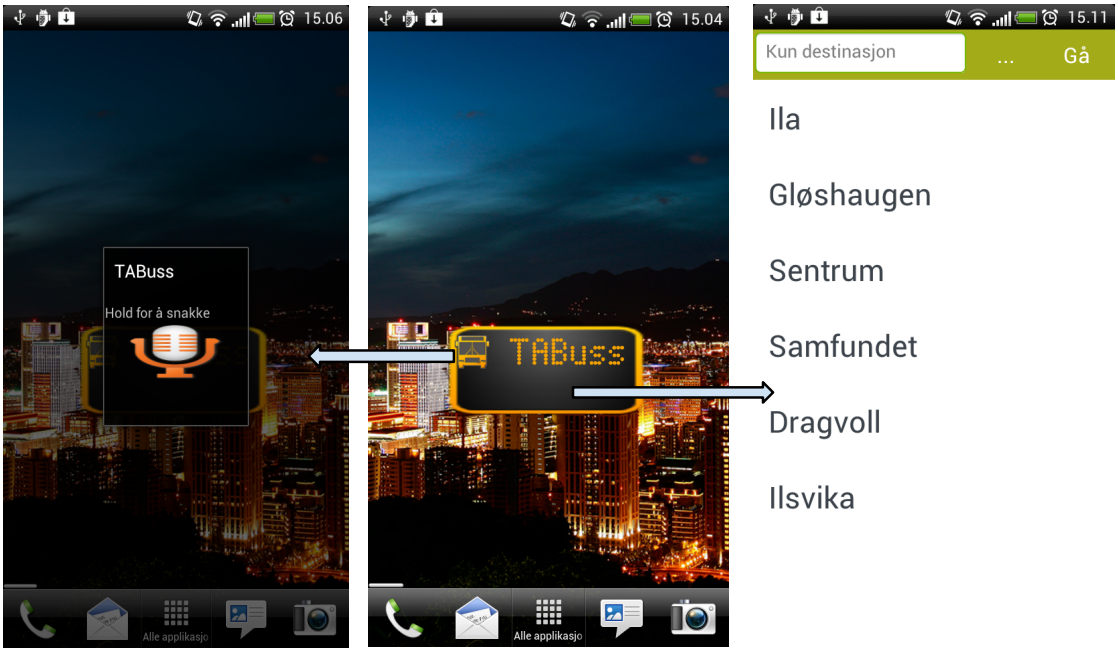

Figure 7.15: Screenshots of the TABuss widget

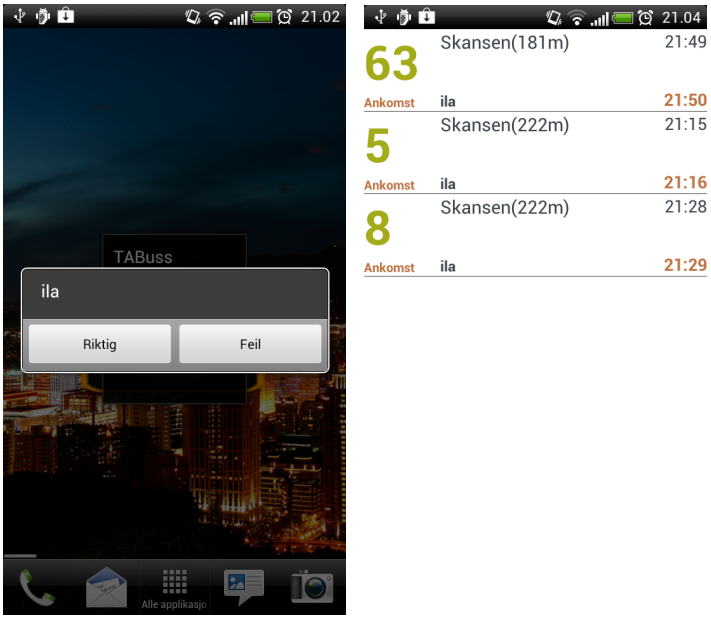

Figure 7.16: The TABuss widget with calculated route suggestions

Chapter 8

Discussion

This chapter first discusses the results, before TaleTUC's research questions and goals are reviewed. Finally, a conclusion is provided.

8.1 The ASR Module

The achieved WERs indicated that the steps performed to make the ASR more robust worked. A low WER was achieved when the clean *acoustic model* was tested indoors, but this highly increased when test sets containing audio recorded outdoors were used. The introduction of the mixed *acoustic model* showed improved results. This was expected, as this model was more diverse.

The introduction of *Sphinx-4*'s *preemphasizer*, *speechClassifier* and *Wiener filter* resulted in further improvements of the WER. The threshold tests showed that the system performed best with a low value set. This meant that a smaller volume difference was required for delineating the foreground noise from the background noise. The results from the *Wiener filter* tests showed significant lower WERs for all of the three test sets (indoors, outdoors, mixed). This showed that even though the amount of available training data was limited, the filter was able to give good estimations. Optimally, an adaptive threshold should be used for the *speechClassifier*, to better cope with noise, but this is not an option in *Sphinx-4*'s functionalities.

The prototype comparisons of *Sphinx-4* and *PocketSphinx* led to expected results, regarding the need for free storage space in *PocketSphinx*. Training with a larger vocabulary and a more robust *acoustic model* than in the developed prototype, can lead to storage space problems. The only real advantage of the *PocketSphinx* solution was that it could process speech offline. For the domain in which TaleTUC is designed to operate, the TABuss client will need to use some data traffic in queries to MultiBRIS. The focus area for TaleTUC was therefore

not to eliminate the data traffic, but to minimise it. The solution, to extract cepstrums, reduced the size of the audio files needed to be sent to the server (see Table 7.11). The size reductions were large enough for TaleTUC to justify the extra time consumption of the extraction process.

8.2 The BSF Module

In the first ASR tests where no filters were used, the BSF module optimised the WERs in all of the tested cases. It also did so in the tests conducted with a *Wiener filter*. These results display the strength of the BSF module, where it is able to map OOV sentences to the correct bus stop names. However, a higher WER was achieved with the BSF and the test set recorded outdoors, compared to the BSF test with the same test set, without a *Wiener filter* (see Table 7.7). This shows that the BSF does not always give large optimisations, even though all OOV sentences are mapped to bus stop names, and also that the size of the optimisations is not fixed.

Though it still needs further testing, the results showed that the BSF provided improved results for the ASR module. For all of the performed tests it was able to replace the OOV words. Combined with the ASR module, it reduced the WERs for all of the tests, compared to the ASR module alone.

8.3 The CBR Module

As can be seen from the CBR results (see section 7.2), there were three parameters tested for their impact on the accuracy of the CBR system. The three parameters were: *Time Variance*, *Distance Variance* and *Noise*. When looking at the CBR results, it is clear that a large amount of noise gave the least accurate CBR solutions. From the practical test (”Natural Scenario”) where only the "core cases" were tested, the system did not crumble until there was over 50% chance that each case was a faulty case. This indicates that the system can hold up against noise around a user’s "core cases". A typical core schedule for a user would for instance be: 06:00 at home and wants to go to work, 16:00 at work and wants to go home and 18:00 at home and wants to go to spear fishing practice. From the 2x10-fold cross validation test, it can seem as this was not the case, since the percent of correct guesses initially seems to drop linear with the amount of noise. What one needs to consider here is that cross validation picks 10% of the test set and tests this against the remaining 90%. If one, or more, of the cases in the 10% are cases where the destination has been altered to a faulty destination, they will be counted as a failed tests even if the CBR system guesses the

"correct" answers for them.

The *Time Variance* and *Distance Variance* parameters seemed to have no impact on the system until they made cases overlap in either time or distance. For instance, both in the 2x10-fold cross validation test and the "Natural Scenario" test, the score fell below 100% at the point where the distance variances exceeded about 2600 metres. This was the distance between the closest core cases.

The tests indicated that the CBR-module works as intended. It works well with variances in the user's schedule, variations in locations and variations in time.

8.4 The Combined Modules

It was shown in section 7.3 that combining the three modules improved the WERs for the test sets. It was also shown that the use of a *strong context* reduced the WER of one of the test sets with as much as 18 %. This happened as the computation was more affected by context data. When looking at these results, there are two important facts. The vocabulary is very domain specific. There is a relatively small set of possible sentences, which is exploited by the BSF to improve the result. The other fact is that the *strong context* depends heavily on the CBR module to produce the correct result. For the CBR module to produce a good result, a set of cases has to exist for the user, which means that the system depends on regular use. The first time a new user accesses the system this will not be the case. An extension could be to client-side ask for the user's daily bus travel schedule, the fist time the system is used.

The results achieved with the combined modules, with and without a *strong context*, indicate that *context-awareness* through CBR is a suitable technology to combine with ASR. The use of a *strong context* lead to the best results, but as mentioned earlier, a case base is necessary. A dynamic switch could be implemented for future use, where the size of the trained case base determines whether or not a *strong context* is used. The combined modules without a *strong context* still provided improved results, and could be used until a sufficient case base is present.

8.5 The Training Application

The training application fulfilled the important purpose of collecting training data from the environment the TaleTUC client is designed to operate in. Volunteers that recorded audio had two choices: either use the application on one of

the project's devices, or use it on their own devices. By doing so, less equipment was necessary, and the recording could take place anywhere. Since the audio files were sent to a server, there were minimal storage requirements. The use of a server also helped in the structuring of files, as the files were automatically fit into a hierarchy. The disadvantage with this approach was the unreliability of the network connection, when moving between WiFi and 3G. This led to incidents where it was necessary to re-record several of the bus stop names.

8.6 The TaleTUC Client and Client Widget

The TaleTUC client and the TaleTUC client widget illustrate the usage of ASR in a bus route information domain, where Android has been used as the target platform. The data traffic was reduced with the client-side cepstrum computations, as discussed in section 8.1. The implementation of this functionality was greatly simplified with the use of the *Sphinx-4* framework. The necessary framework components are portable to the Android platform through Java, and were directly implemented.

The challenge on user frustration was described in section 2.5.3. Both the TaleTUC client and the TaleTUC client widget offer user verification of the ASR result. Whether users find this beneficial, or just a waste of time, have to be tested. To verify the ASR result requires additional user interaction, which might not be desirable if one is occupied with something else. On the other hand, unneccesary data traffic and the time use of MultiBRIS queries with the wrong ASR result, are avoided. For the time being, because of the time use of MultiBRIS, user verification is seen as an important measure to minimise the user frustration.

The other approach to minimise the user frustration, was to only allow two ASR attempts (as described in section 7.4.1). This approach will also need testing to uncover areas that need improvement, as for instance the number of tries a user should get. But is for the time being also seen as important.

8.7 Research Questions and Goals

The following sections list the research questions and goals, and review their fulfillment.

8.7.1 Research Question 1

Which ASR technologies are well suited for Android devices and the task at hand? The performed research identified several commercial and non-commercial ASR technologies. Each of these were evaluated on whether or not they were suitable for TaleTUC, and suitable for Android devices. The best candidate found was *Sphinx-4*, which became the chosen ASR engine for TaleTUC.

8.7.2 Research Question 2

Where is it most desirable to do the different parts of the ASR? On the device or on a server? Given the researched technologies and the space requirements of client-side ASR, it was most desirable to perform the ASR on a server. Client-side extraction of cepstrums was also implemented, which reduced the data traffic requirements.

8.7.3 Research Question 3

Can context-awareness through CBR optimise the performance of an ASR system? WER improvements were achieved for all of the test sets when *context-awareness* through CBR were combined with ASR.

8.7.4 Goal 1

Develop a prototype system based on the results from research question 1 and 2. The developed prototype is based on the technology result from research question 1, which was *Sphinx-4*. From research question 2, a client-server architecture was used. Cepstrum extraction of the recorded speech was also implemented on the test devices, and runs showed that this led to reduced data traffic.

The prototype is implemented to recognise 50 bus stop names in Trondheim. The lowest achieved WERs with the use of a *Wiener filter* and the BSF were 22 % for the test set recorded outdoors, 0 % for the test set recorded indoors and 4 % for the mixed test set (recorded both indoors and outdoors). It has a modular design, and future expansions of the vocabulary or adaptation to other domains is possible without having to make any architectural changes.

8.7.5 Goal 2

Develop modules for context-awareness based on research question 3, and integrate them with the prototype. The BSF and CBR modules were created. Tests were conducted with the ASR module combined with the BSF module, and the ASR module combined with both the BSF and CBR modules. This yielded good results, as described in the Result section (section 7.1).

8.7.6 Goal 3

Integrate the prototype with existing parts of the FUIROS-project. TABuss now operates as a TaleTUC client. TABuss' main functionality can use the ASR result as input for querying route suggestions through MultiBRIS.

A widget has also been created, which originally was not within this goal's scope. This widget offers quicker access to the ASR functionality together with TABuss' main functionality.

8.8 Conclusion

We are satisfied with the development process, including the learning of new theory and technologies. There were no major problems during the development, and the performed research created the foundation for a functional prototype.

The *acoustic model* in the ASR module should both be trained on more speakers, and trained on more bus stop names. The implemented *Wiener filter* would also benefit from this, as it would improve the estimations of clean speech. However, it was not easy within the development time frame to get many participants to record audio. Because of the prototype's modular design, expanding the ASR module's *dictionary* or adding more training data can easily be done, if enough people are available. This also applies if the system is to be adapted towards another domain than bus route information. The Java program developed to automate the steps for creating the *language model* only needs an updated text file containing the added entries, to create a new *language model*. The same is for the building of the *acoustic model*. Given an updated *dictionary* and recorded utterances, the program runs the necessary scripts, and creates an *acoustic model*.

Another advantage that comes with the modularity is that future TaleTUC clients can use the same server functionalities. This is the benefit of a client-server architecture. Such clients can be based on any operating system, as long as audio can be recorded and sent over HTTP.

There was not much focus on the user interface for the TaleTUC client. To use the ASR functionality one has to access one of the TABuss home screen menu options. We are however satisfied with the client widget. In our opinion, the widget increases the chances that the ASR functionality will be used, because it is easy to access from the home screen. Its design is created with simple access in mind, and an optimal user interface in the client could be similar to this.

// Ha med noe om cbr, evt combined

Section 2.6.2 describes the *CU-Move* and the *CU Communicator*. It was mentioned that the results from *CU-Move*'s use of *context-awareness* could be compared to TaleTUC's use. The *CU-Move* system achieved 3.2 % WER for digits, in a noisy environment. The most comparable test set used in TaleTUC is the set that contains audio recorded outdoors, where runs showed a WER of 4 %, with the use of a *strong context*. However, it is optimistic to assume that TaleTUC in real-life provides only a 0.8 % higher WER than *CU-Move*. *CU-Move* is more robust, and used a training set containing 4000 utterances, and a test set containing 500 utterances. In addition, it was designed for in-vehicle ASR, which means that the two systems are not tested in the same conditions. Still, TaleTUC's results show that the use of *context-awareness* has been successful. As *CU-Move* also achieved good results, it is reasonable to conclude that there are multiple ways to use *context-awareness* to optimise ASR.

Let's go was another related system. Raux et al. (2005) achieved a WER of 68 %, which is substantially higher than in TaleTUC. One of the main reasons for the high WER is errors related to the structuring of senteces. Despite the fact that *Let's go* uses a larger vocabuarly, which gives more error sources, we claim that the TaleTUC approach provides better results. First of all, TaleTUC does not use a calling interface, and can utilise the smartphone's built-in sensors such as GPS. The client only requires one input, which avoids the wrong structuring of sentences. Secondly, TaleTUC uses context data through CBR to optimise the process, and to avoid OOV words. And because TABuss is used as a TaleTUC client, the user has other options for getting route suggestions, if the ASR cannot provide the correct ones. This, and actions for handling failed recognitions, were designed to cope with the challenge identified by Raux et al. (2005), on user frustration.

The goals for this project were to research ASR for mobile devices, and create a prototype that combines ASR with *context-awareness* through CBR. The prototype showed successful results, and all of the goals were fulfilled. TaleTUC can also be viewed as a foundation for future work, as there still is work that needs to be done. Typical next steps, which are further described in chapter 9, are to perform user testing, expand the vocabulary with more bus stop names and

extend onto other domains than bus route information.

Chapter 9

Future Work

This chapter identifies the future work for TaleTUC, and the future work for FUIROS and the FUIROS related technologies.

9.1 TaleTUC

First of all, TaleTUC and the TaleTUC client should be user tested. TABuss (Marcussen and Eliassen, 2011) received feedback that indicated that it suited the needs of bus travellers. Since TABuss as a TaleTUC client uses the existing route suggestion functionalities, it would be interesting to get updated user feedback. TaleTUC should also be tested and compared against the existing *Buster* (Hartvigsen et al., 2007) system. Performed tests could give answers to whether the users prefer ASR with complete sentences, as in *Buster*, or the setup that TaleTUC and the TaleTUC client use. Testing of the performance of TaleTUC's *Sphinx-4* implementation could also be done, by creating prototypes based on the technologies that were not chosen. Models could be created in *HTK*, and both *HTK* and *Julius* decoders could be used to test the accuracy.

As mentioned in section 8.8, TaleTUC's *acoustic model* should be trained on more speakers (and dialects) and bus stop names. This is essential if a TaleTUC client such as TABuss is to be released with ASR as a functionality. An interesting idea is to let the users each have their own *acoustic model* residing on the server. If the ASR module, the BSF module and the CBR module should fail on a spoken utterance, this audio could be used to train the *acoustic model* for a user provided bus stop name. In this way, over time, each user would have an adapted *acoustic model*. The challenge is space requirements. To let each user have his or her own *acoustic model* will naturally occupy more space than having only one for all, and the space requirements would increase with the number of users.

The latest *acoustic model* in TaleTUC is trained on ten speakers. Each of the training sets contain 50 recordings (one for each bus stop name), where each recording has a length of approximately two seconds. This means that each person recorded 100 seconds of audio for a training set. To train TaleTUC on all of the bus stop names in the Trondheim area, which is approximately 2700, a training set would contain 5400 seconds of audio, or 90 minutes. These calculations do not account for time spent starting the training application, navigating the training application or any erroneous recordings. How much these factors affect the total time use is individual, but the actual time spent for the process of creating a training set is guaranteed to be higher than just the total length of the recordings. It is important to reduce this additional time use, as it can effect the participation of volunteers for creating training sets. A re-design of the training application may be a solution, or to re-think the training process as a whole.

The Java program that automates the building steps for the *language model* and the *acoustic model* should be designed to support the Windows operating system. This can either be done through implementing an operating system detector, or by re-designing the programs entirely.

The existing prototype should be made more robust. This project has investigated different built-in *Sphinx-4* approaches for robust ASR, but other approaches need to be researched. This can be different noise cancelling algorithms or filters. *Sphinx-4*'s XML configuration is plug-in based, such that algorithms or filters can be integrated into a module, and plugged into the configration front-end. An example is to develop a VAD module with an adaptive threshold (Jiang et al., 2010), which could replace *Sphinx-4*'s *speechClassifier*.

To further reduce the data traffic for the TaleTUC client, different codecs could be investigated. There are a number of alternatives available, where Speex[1] and Ogg Vorbis[2] are two popular ones. However, because the data traffic already has been heavily reduced caused by the extraction of cepstrums, the use of compression functionalities have to keep the run time in mind. A compression algorithm that requires as little as a second of computation time, should provide a great reduction of audio file sizes if it is to be considered. In addition, if a lossy compression algorithm is used, information is lost.

The lack of effort put into the user interface of the TaleTUC client was mentioned in section 8.8. Future work could start with the user interface in the TaleTUC client widget as a foundation, and conduct user tests with this. An intuitive user interface is crucial for the user experience, and can actually be a contributing factor to whether the functionality will be used or not.

Other work includes to develop TaleTUC clients based on other technolo-

[1] `http://www.speex.org/`

[2] `http://www.vorbis.com/`

gies than Android, to demonstrate TaleTUC's modularity. An example is to start with the *hybrid* MultiBRIS client, which is developed in JavaScript[3]. This client provides the same functionalities as TABuss, and for it to operate as a TaleTUC client would only require the implementation of methods for recording and sending audio.

A possibility that concerns the entire TaleTUC system (including Engell (2012)'s work), is route guidance, similar to the *TravelMan* system (Turunen et al., 2007). The application of ASR for route guidance has also been researched by Komatani et al. (2003), who developed a dialogue system, tested in the city of Kyoto. A route guidance functionality in TaleTUC would be beneficial for tourists and people not familiar with the different places in Trondheim. A basic implementation for the TABuss client is to use speech synthesis for route suggestions. The provided suggestions already contain all of the needed information, and the integration of speech synthesis is only a matter of fitting this information into sentences.

In relation to CBR there are many improvements that could be explored. One of these is to make the SQL queries more efficient by adding a max range to the queries. Another improvement could be to implement a functionality that lets the user reset his or her CBR case base. This could be handy, say if one moves, or changes workplace. Another solution to this would be to give older cases less credibility. In this way, older cases would "fade out" of the system over time. This could also help to prune the case database. If a case has "all faded out" in the way that it does no longer contribute to a solutions' score, it could be removed from the database. This would help the case database to maintain a small size, and by that maintaining short query times.

[3] `http://www.w3schools.com/js/`

9.2 FUIROS and FUIROS Related Technologies

The following sections describe the future work for FUIROS in general, TABuss, MultiBRIS and BusTUC.

9.2.1 Geographical Expansion of FUIROS and Standards

An idea for future work for FUIROS, is to add support for other cities in Norway, and use a single system to provide public transportation information for the entire country. Then, a single client application could access route information based on the mobile device's location.

A challenge for an effective expansion is the need for standards. It would aid the development if all of the bus agencies in Norway used the same standards for sharing routes and real-time data. Norway's largest bus agency, Trafikanten AS [4], already uses such a standard. This standard, which is called SIRI[5], is used for the distribution of real-time data. It is an XML protocol that allows distributed computers to exchange real-time information about public transport services and vehicles.

Through a JSON-API Trafikanten AS has made the *StopMonitoring (SM)* part of SIRI available for public use. The Stop Monitoring section is described by the SIRI standard as follows:

> *The Stop Services (Stop Timetable and Stop Monitoring) The Stop Timetable (ST) and Stop Monitoring services (SM) provide stop-centric information about current and forthcoming vehicle arrivals and departures at a nominated stop or Monitoring Point, typically for departures within the next 20-60 minutes for display to the public. The SM service is suited in particular for providing departure boards on all forms of device.*

SIRI is already in use by Trafikanten, and therefore it represents a good example of what could be a national standard for sharing real-time public transport information. For the notion of a single system, this could be the first step towards achievement.

However, the biggest challenge for such standards is probably not technical, but rather political and financial. An approach that avoids the distributed standardisation challenge could be constructed by absorbing the existing transportation agency systems one-by-one. This system would effectually become

[4] `www.ruter.no`

[5] `http://www.kizoom.com/standards/siri/`

the mediation layer that creates the standard, seen from an application developer's point-of-view. This would also increase the amount of work needed to expand the system substantially, compared to expanding a system based on standards. The advantage to this approach would be that such a system could establish a position of power in relation to public transport data sharing. It is reasonable to believe that a system that has a standard way to communicate route data for an entire country, would become vastly popular in the development community. By providing the "back-end" to "front-end" mediation layer for the majority of available public transportation client-applications available, one would be in a position of power. This is an advantage that could be used to encourage the use of standards such as SIRI, among the public transportation agencies.

9.2.2 TABuss

The following sections first identify possibilities with the new smartphone technologies. Future work involving *context-awareness* is then described. Finally, suggestions for future extensions to TABuss are provided.

New Smartphone technologies

Based on the experiences with widget development for the TaleTUC client widget, a widget could be created for TABuss. This widget could provide information such as real-time passings of buses for the closest bus stop to the user's location. Touch events could trigger the widget itself to provide som information, or trigger the start-up of TABuss.

Another interesting field is Near Field Communication (NFC) (Ylinen et al., 2009), and the use of this technology in mobile applications (Sánchez et al., 2012). A usage in TABuss could be to detect Radio Frequency Identification (RFID) (Ngai et al., 2008) tags that have been integrated into in every bus stop. When the user is close enough to a bus stop, the application could trigger the display of the next passing buses. RFID tags integrated into bus stops could also be used for speech synthesis purposes. Blind people, or others with bad eyesight, could benefit from a functionality where the system reads out loud the next passing buses, when they approach a bus stop.

A new implementation that involves AtB, is the purchase of bus tickets. It is possible to buy tickets through a service provided by AtB, by sending a text message to 2027 (Norwegian number), and specifying the type of ticket (adult, child, military, etc). This sending of a text message could be triggered by the user approaching the bus stop. It should be integrated into already existing functionalities, to avoid unnecessary sending of text messages. An example

is when the user has performed a query to MultiBRIS, and has received route suggestions. The user could then select the suggestion he or she wants to use, an action that alerts the RFID reader to start the SMS service when the user approaches the selected departure bus stop.

Context-Awareness

An extension involving *context-awareness* for TABuss is to use more sensors than only the location sensor, which has been done by Raento et al. (2005). Their system uses four sensors: location, user interaction, communication behaviour and physical environment. This means that besides from location information, their system monitors: what actions the user performs, calls and text messages and surrounding devices.

For TABuss, this sensor information could be used to introduce *context-awareness* to the user interface. The age differences between potential target users is large, and an adaptive user interface could be a solution. The user interface could through sensors track the user's actions, register some trends and then adjust visibility and availability accordingly. An example is to track the usage of the ASR module. If it is an often used feature, access to it could be made quicker.

The tracking of user trends could also be used to perfect route suggestions. People of different ages have different levels of mobility, and have different walking speeds. This has been addressed by Vieira et al. (2011), in their proposed system *UbiBus*. *UbiBus* considers different people's and vehicle's mobility, and other factors than can affect a bus departure. An interesting idea is for AtB to contribute to such functionalities in order to improve route suggestions. Buses have installed cameras, and could be used to monitor how crowded a bus is. This could prove beneficial for handicapped people, or people with small children, who need seats or at least clear floor area.

Another suggestion is to use *context* through calendar information, by monitoring scheduled appointments. When an appointment is approaching, the user could be prompted with a query suggestion. Khalil and Connelly (2005) stated that it is an inevitable fact that people's actions not always mirror their intentions. Even though an appointment has been scheduled, the user is not guaranteed to attend. TABuss queries should therefore not be run automatically in this case, only a query suggestion should be prompted. Automatic query runs could cause unnecessary data traffic when the user has chosen not to attend a scheduled appointment, or has chosen another form of transportation.

A challenge with introducing *context-aware* extensions is privacy. If such information is to be stored on a server, a secure login mechanism is necessary. TaleTUC uses the device IDs of the smartphones to separate users, which is a sufficient solution when non-sensitive data is stored. To introduce the factors

proposed by Raento et al. (2005) will either require that these factors are stored on the device, or that a secure storage functionality is created server-side. With server-side storage and a login mechanism, it is important to minimise the user requirements. Users may reject an application that requires too many involvements, when they want to get a quick route suggestion.

Future Extensions of TABuss

A future extension could be to integrate TABuss into a tourist application. The *Trondheim Guide*[6] is an intelligent travel guide which already provides some bus route information. This information is limited, and no information on arrival/departure times was found during testing. Another alternative is *City Explorer*[7], which is a framework for city exploration. In relation to TaleTUC, tourist information could be a domain to extend the ASR functionality to cover. Then, TABuss as a TaleTUC client with integrated *City Explorer* functionalites could use this.

9.2.3 MultiBRIS

Flinn et al. (2002) developed a system that dynamically decides whether to perform server-side or client-side computations. These decisions are based on monitored resource usage both on the server and the client. This functionality could be implemented for MultiBRIS, and prevent delays when the MultiBRIS server is busy, which can be caused by a high traffic load. In those situations, the clients should do the necessary operations instead of relying on MultiBRIS. Clients such as TABuss must have functionality that calculates route suggestions, and allow for queries to be sent to BusTUC and AtB's real-time system. This puts extra computational pressure on the client, but facilitates a solution that can provide route suggestions with and without the involvement of MultiBRIS.

Flinn et al. (2002) also describes the idea to let the client learn what is best practise in the different situations, when taking into account factors such as low battery power. The client could monitor the resource usage for performed operations over time, and learn which tasks to compute client-side and which to compute server-side. Experiences gained after each operation could be stored as cases, and a CBR functionality could be used for retrievement.

[6] `www.trondheim.no/app`

[7] `http://www.sintef.no/Projectweb/UbiCompForAll/Results/Software/City-Explorer/`

9.2.4 BusTUC

Future work on BusTUC includes the research of other intelligent route information solutions. Because BusTUC is the only available candidate in Trondheim, there are no systems to compare it with. One specific task would be to do research on similar systems found outside of Trondheim, and develop comparable prototypes. If research shows that BusTUC is the best solution, a goal could be to establish it as a standard for bus route information in Norway. This standard, together with the SIRI standard, could then be two of the building blocks of a common standard, for the exchange of transportation information.

Another option is to expand BusTUC and the concept of a natural language route information system outside of Trondheim, to cities of different sizes and number of inhabitants.

Chapter 10

Acknowledgements

We would like to thank our supervisors Rune Sætre and Björn Gambäck for their guidance. We would also like to thank the people who contributed to the recording of the training sets: André Christoffer Andersen, Lars Moland Eliassen, Trond Bøe Engell, Marita Gjerde, Stian Pedersen Kvaale, Stian Mikelsen, Morten Stornes and Anne Olga Syverhuset.

Christoffer would like to thank Liverpool Football Club. We have had ups and downs throughout our relationship, but it is good to see that we both now are ready for a new chapter to begin.

Bibliography

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59.

Alewine, N., Ruback, H., and Deligne, S. (2004). Pervasive speech recognition. *Pervasive Computing, IEEE*, 3(4):78 – 81.

Andersstuen, R. and Engell, T. (2011). Multibris: - a multiple-platform approach to the ultimate bus route information system for mobile devices. Technical report.

Arakawa, T., Tsujikawa, M., and Isotani, R. (2006). Model-based wiener filter for noise robust speech recognition. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, page I.

Baker, J. (1975). The dragon system–an overview. 23:24–29.

Ballinger, B., Allauzen, C., Gruenstein, A., and Schalkwyk, J. (2010). On-demand language model interpolation for mobile speech input. *INTERSPEECH-2010*, pages 1812–1815.

Bangalore, S. and Johnston, M. (2000). Integrating multimodal language processing with speech recognition. volume 2 of *In ICSLP-2000*, pages 126–129, Florham Park, NJ, USA. AT and T Labs Research, Shannon Laboratory.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics*, 41(1):pp. 164–171.

Bazzi, I., Glass, J., and Smith, A. C. (2000). Modeling out-of-vocabulary words for robust speech recognition.

Bolt, R. A. (1980). Put-that-there: Voice and gesture at the graphics interface. *SIGGRAPH Comput. Graph.*, 14(3):262–270.

Box, G. E. and Tiao, G. C. (1992). *Front Matter*, pages i–xviii. John Wiley and Sons, Inc.

Chen, J., Benesty, J., Huang, Y., and Doclo, S. (2006). New insights into the noise reduction wiener filter. *Audio, Speech, and Language Processing, IEEE Transactions on*, 14(4):1218 –1234.

Clarkson, P. and Rosenfeld, R. (1997). Statistical language modeling using the cmu-cambridge toolkit. In *PROCEEDINGS EUROSPEECH*, pages 2707–2710.

Cosi, P. and Nicolao, M. (2009). Connected digits recognition task: Istc-cnr comparison of open source tools.

Delaney, B., Jayant, N., and Simunic, T. (2005). Energy-aware distributed speech recognition for wireless mobile devices. *Design Test of Computers, IEEE*, 22(1):39 – 49.

Deng, L. and Huang, X. (2004). Challenges in adopting speech recognition. *Commun. ACM*, 47(1):69–75.

Deng, L. and O'Shaughnessy, D. (2003). *Speech processing: a dynamic and optimization-oriented approach*. Marcel Dekker, 1 edition.

Dudley, H. (1939). The vocoder. 17:122–126.

Dumas, B., Lalanne, D., and Oviatt, S. (2009). Multimodal interfaces: A survey of principles, models and frameworks. In Lalanne, D. and Kohlas, J., editors, *Human Machine Interaction*, volume 5440 of *Lecture Notes in Computer Science*, pages 3–26. Springer Berlin / Heidelberg.

Ehlen, P. and Johnston, M. (2012). Multimodal interaction patterns in mobile local search. In *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*, IUI '12, pages 21–24, New York, NY, USA. ACM.

Engell, T. (2012). Taletuc: Text-to-speech and other en- hancements for an existing bus route information systems. Master's thesis, NTNU.

Evermann, G. (1999). Minimum word error rate decoding.

Flinn, J., Park, S., and Satyanarayanan, M. (2002). Balancing performance, energy, and quality in pervasive computing. In *In Proceedings of the 22nd International Conference on. Distributed Computing Systems*, pages 217–226.

Floreano, D. and Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. The MIT Press.

Forman, G. H. and Zahorjan, J. (1994). The challenges of mobile computing. *Computer*, 27(4):38–47.

Forney, G.D., J. (1973). The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268 – 278.

Gorin, A., Parker, B., Sachs, R., and Wilpon, J. (1996). How may i help you? In *Interactive Voice Technology for Telecommunications Applications, 1996. Proceedings., Third IEEE Workshop on*, pages 57 –60.

Hansen, J. and Clements, M. (1991). Constrained iterative speech enhancement with application to speech recognition. *Signal Processing, IEEE Transactions on*, 39(4):795–805.

Hansen, J., Zhang, X., Akbacak, M., Yapanel, U., Pellom, B., Ward, W., and Angkititrakul, P. (2005). Cu-move: advanced in-vehicle speech systems for route navigation. *DSP for in-vehicle and mobile systems*, pages 19–45.

Hartvigsen, O., Harborg, E., Amble, T., and Johnsen, M. (2007). Marvina - a norwegian speech centric, multimodal visitors guide. In *NODALIDA 2007 Proceedings (The 16th Nordic Conference of Computational Linguistics)*.

Hazas, M., Scott, J., and Krumm, J. (2004). Location-aware computing comes of age. *Computer*, 37:95–97.

Huang, X., Acero, and Hon, H. (2001). *Spoken language processing: a guide to theory, algorithm, and system development*. Prentice Hall PTR.

Huang, X., Alleva, F., Hon, H.-w., Hwang, M.-y., and Rosenfeld, R. (1992). The sphinx-ii speech recognition system: An overview. *Computer, Speech and Language*, 7:137–148.

Huggins-Daines, D., Kumar, M., Chan, A., Black, A., Ravishankar, M., and Rudnicky, A. (2006). Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1.

Itakura, F. (1975). Minimum prediction residual principle applied to speech recognition. ASSP-23:57–72.

Jiang, W., Lo, W. K., and Meng, H. (2010). A new voice activity detection method using maximized sub-band snr. In *Audio Language and Image Processing (ICALIP), 2010 International Conference on*, pages 80 –84.

Juang, B. and Rabiner, L. R. (2005). Automatic speech recognition - a brief history of the technology development.

Juang, B.-H., Chou, W., and Lee, C.-H. (1997). Minimum classification error rate methods for speech recognition. 5:257–265.

Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2 edition.

Khalil, A. and Connelly, K. (2005). Improving cell phone awareness by using calendar information. In Costabile, M. and Paternò, F., editors, *Human–Computer Interaction – INTERACT 2005*, volume 3585 of *Lecture Notes in Computer Science*, pages 588–600. Springer Berlin / Heidelberg.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International joint Conference on artificial intelligence*, volume 14, pages 1137–1145. LAWRENCE ERLBAUM ASSOCIATES LTD.

Komatani, K., Ueno, S., Kawahara, T., and Okuno, H. G. (2003). Flexible guidance generation using user model in spoken dialogue systems. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 256–263, Stroudsburg, PA, USA. Association for Computational Linguistics.

Kühnel, C., Westermann, T., Weiss, B., and Möller, S. (2010). Evaluating multimodal systems: a comparison of established questionnaires and interaction parameters. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI '10, pages 286–294, New York, NY, USA. ACM.

Kumar, A., Tewari, A., Horrigan, S., Kam, M., Metze, F., and Canny, J. (2011). Rethinking speech recognition on mobile devices.

Le, K.-F. (1988). *Large-vocabulary speaker-independent continuous speech recognition: The Sphinx system*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Lee, A. and Kawahara, T. (2009). Recent development of open-source speech recognition engine julius.

Lee, C.-y., Glass, J. R., and Ghitza, O. (2011). An efferent-inspired auditory model front-end for speech recognition. In *INTERSPEECH*, pages 49–52. ISCA.

Lippmann, R. (1997). Speech recognition by machines and humans. *Speech communication*, 22(1):1–15.

Liu, F.-H., Stern, R. M., Huang, X., and Acero, A. (1993). Efficient cepstral normalization for robust speech recognition. In *Proceedings of the workshop on Human Language Technology*, HLT '93, pages 69–74, Stroudsburg, PA, USA. Association for Computational Linguistics.

Lowerre, B. (1976). *The harpy speech recognition system.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA. AAI7619331.

Ma, T., Kim, Y.-D., Ma, Q., Tang, M., and Zhou, W. (2005). Context-aware implementation based on cbr for smart home. In *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005), IEEE International Conference on*, volume 4, pages 112 – 115 Vol. 4.

Maier, V. and Moore, R. (2009). The case for case-based automatic speech recognition. In *Tenth Annual Conference of the International Speech Communication Association*.

Marcussen, C. and Eliassen, L. M. (2011). Tabuss: - an intelligent smartphone application. Technical report.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing. `http://www.nist.gov/itl/cloud/index.cfm`.

Miguel, A., Buera, L., Lleida, E., Ortega, A., and Saz, O. (2007). On-line feature and acoustic model space compensation for robust speech recognition in car environment. In *Intelligent Vehicles Symposium, 2007 IEEE*, pages 1019 –1024.

Molau, S., Pitz, M., Schluter, R., and Ney, H. (2001). Computing mel-frequency cepstral coefficients on the power spectrum. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, volume 1, pages 73 –76 vol.1.

Muda, L., Begam, M., and Elamvazuthi, I. (2010). Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *CoRR*, abs/1003.4083.

Narayanan, A., Zhao, X., Wang, D., and Fosler-Lussier, E. (2011). Robust speech recognition using multiple prior models for speech reconstruction. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 4800 –4803.

Ngai, E., Moon, K. K., Riggins, F. J., and Yi, C. Y. (2008). Rfid research: An academic literature review (1995-2005) and future research directions. *International Journal of Production Economics*, 112(2):510 – 520.

NIMA (2000). Department of defense world geodetic system 1984, its definition and relationships with local geodetic systems. Technical report, National Geospatial-Intelligence Agency.

Odell, J., Kershaw, D., Ollason, D., Valtchev, V., and Whitehouse, D. (1999). *The HAPI book. A description of the HTK Application Programming Interface*. Entropic Ltd, 1.4 edition.

Oviatt, S. (1999). Ten myths of multimodal interaction. *Commun. ACM*, 42(11):74–81.

Pedersen, P. (1965). The mel scale. *Journal of Music Theory*, 9(2):pp. 295–308.

Pellom, B. and Hansen, J. (1998). An improved (auto: I, lsp: t) constrained iterative speech enhancement for colored noise environments. *Speech and Audio Processing, IEEE Transactions on*, 6(6):573–579.

Pellom, B., Ward, W., Hansen, J., Cole, R., Hacioglu, K., Zhang, J., Yu, X., and Pradhan, S. (2001). University of colorado dialog systems for travel and navigation. In *Proceedings of the first international conference on Human language technology research*, pages 1–6. Association for Computational Linguistics.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286.

Rabiner, L. R., Levinson, S. E., Rosenberg, A. E., and Wilpon, J. G. (1979). Speaker independent recognition of isolated words using clustering techniques. ASSP-27:336–349.

Raento, M., Oulasvirta, A., Petit, R., and Toivonen, H. (2005). Contextphone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4:51–59.

Ramirez, J., Gorriz, J., and Segura, J. (2007). Voice activity detection. fundamentals and speech recognition system robustness. *System*, (June):1 –22.

Raux, A., Langner, B., Black, A. W., and Eskenazi, M. (2003). Let's go: Improving spoken dialog systems for the elderly and non-natives. In *in Eurospeech03*.

Raux, A., Langner, B., Bohus, D., Black, A. W., and Eskenazi, M. (2005). Let's go public! taking a spoken dialog system to the real world. In *in Proc. of Interspeech 2005*.

Rozman, R. and Kodek, D. M. (2006). Abstract using asymmetric windows in automatic speech recognition.

Rudnicky, A. I., Rudnicky, E. I., Bennett, C., Black, A. W., Chotomongcol, A., Lenzo, K., Oh, A., and Singh, R. (2000). Task and domain specific modelling in the carnegie mellon communicator system. In *In ICSLP200*, pages 130–134.

Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

Salonen, E.-p., Turunen, M., Hakulinen, J., Helin, L., and Prusi, P. (2005). Distributed dialogue management for smart terminal devices.

Sánchez, M., Mateos, M., Fraile, J., and Pizarro, D. (2012). Touch me: A new and easier way for accessibility using smartphones and nfc. In Pérez, J. B., Sánchez, M. A., Mathieu, P., Rodríguez, J. M. C., Adam, E., Ortega, A., Moreno, M. N., Navarro, E., Hirsch, B., Lopes-Cardoso, H., and Julián, V., editors, *Highlights on Practical Applications of Agents and Multi-Agent Systems*, volume 156 of *Advances in Intelligent and Soft Computing*, pages 307–314. Springer Berlin / Heidelberg.

Sarikaya, R. and Hansen, J. (2000). Improved jacobian adaptation for fast acoustic model adaptation in noisy speech recognition. In *Sixth International Conference on Spoken Language Processing*.

Sarosi, G., Mozsary, M., Mihajlik, P., and Fegyo, T. (2011). Comparison of feature extraction methods for speech recognition in noise-free and in traffic noise environment. In *Speech Technology and Human-Computer Dialogue (SpeD), 2011 6th Conference on*, pages 1 –8.

Satori, H., Harti, M., and Chenfour, N. (2007). Introduction to arabic speech recognition using CMUSphinx system.

Schank, R. (1983). *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge University Press.

Sinnott, R. (1984). Virtues of the haversine. *Sky and telescope*, 68:158.

Sriratanaprapahd, S. and Songwatana, K. (2008). Design and implementation of the thai language voice recognition system for the private phonebook service

on the telephony network. In *Communications and Information Technologies, 2008. ISCIT 2008. International Symposium on*, pages 274 –277.

Tan, Z.-h., Dalsgaard, P., and Lindberg, B. (2005). Automatic speech recognition over error-prone wireless networks. *Speech Communication*, 47.

Trutnev, A. and Rajman, M. (2004). Comparative evaluations in the domain of automatic speech recognition. In *International Conference on Language Resources and Evaluation (LREC2004)*, Lisbon, Portugal.

Turunen, M. and Hakulinen, J. (2003). Jaspis – an architecture for supporting distributed spoken dialogues. In *IN EUROSPEECH 2003*, pages 1913–1916.

Turunen, M., Hakulinen, J., and Kainulainen, A. (2006a). Evaluation of a spoken dialogue system with usability tests and longterm pilot studies: Similarities and differences. In *In Proceedings of Interspeech 2006*.

Turunen, M., Hakulinen, J., Kainulainen, A., Melto, A., and Hurtig, T. (2007). Design of a rich multimodal interface for mobile spoken route guidance.

Turunen, M., Hurtig, T., Hakulinen, J., Virtanen, A., and Koskinen, S. (2006b). Mobile speech-based and multimodal public transport information services.

Vaquero, L. M., Rodero-merino, L., Caceres, J., Lindner, M., and Desarrollo, T. I. Y. (2009). A break in the clouds: Towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, pages 50–55.

Vieira, V., Caldas, L. R., and Salgado, A. C. (2011). Towards an ubiquitous and context sensitive public transportation system. *International Conference on UbiMedia Computing*, 0:174–179.

Vincentry, T. (1975). Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, XXIII:88–93.

Walker, W., Lamere, P., Kwok, P., Raj, B., Singh, R., Gouvea, E., Wolf, P., and Woelfel, J. (2004). Sphinx-4: A flexible open source framework for speech recognition. Technical report.

Wilpon, J., Rabiner, L. R., and Bergh, A. (1982). Speaker-independent isolated word recognition using a 129-word airline vocabulary. 23:390–395.

Xu, W., Guo, Y., Wang, B., Wang, X., and Mai, Z. (2005). A noise robust front-end using wiener filter, probability model and cms for asr. In *Natural Language Processing and Knowledge Engineering, 2005. IEEE NLP-KE '05. Proceedings of 2005 IEEE International Conference on*, pages 102 – 105.

Yazgan, A. and Saraclar, M. (2004). Hybrid language models for out of vocabulary word detection in large vocabulary conversational speech recognition. In *Proc. Int. Conf. of Acoustics, Speech, and Signal Processing*, pages 745–748.

Ylinen, J., Koskela, M., Iso-Anttila, L., and Loula, P. (2009). Near field communication network services. In *Digital Society, 2009. ICDS '09. Third International Conference on*, pages 89 –93.

Zaykovskiy, D. (2006). Survey of the speech recognition techniques for mobile devices.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18.

Zhao, Y., Wakita, H., and Zhuang, X. (1991). An hmm based speaker-independent continuous speech recognition system with experiments on the timit database.

Zue, V., Seneff, S., Glass, J., Polifroni, J., Pao, C., Hazen, T. J., and Hetherington, L. (2000). Jupiter: A telephone-based conversational interface for weather information. *IEEE Trans. on Speech and Audio Processing*, 8:85–96.

Appendix

This appendix contains information on how to do future development of TaleTUC.

How To Get Started With Future Development of TaleTUC

The following sections first describe the necessary programs. Instructions are then given on how to test the existing functionalities, and how to build and train models. Finally, information on the TABuss source code, and the TaleTUC server are provided.

Install Sphinx-4, PocketSphinx and Additional Tools

Sphinx-4, *PocketSphinx* and all of the additional tools are available from the *Sphinx* website `http://cmusphinx.sourceforge.net/`. This site also provides the necessary tutorials to get started with development. This includes tutorials on building *language models* and *acoustic models*.

The *acoustic model* tutorial's first step is to create a certain file structure. The file structure used in TaleTUC is found in the *TestTrain* folder, where the two relevant top-level folders are *Acoustic_model* and *Language_model*.

Test the Existing ASR Functionality

Testing of the latest models can be done in the *Lattice* project. The *Lattice* project is a NetBeans[1] project that can be used to run ASR tests locally on a computer. It does not use the CBR module, but tests can be done with and without the BSF. It contains five classes:

[1] `www.netbeans.org`

LatticeMFCC.java is used for ASR on a single cepstrum file. The cepstrum of a wav file is extracted and saved to a separate file, before it is used as input to the recogniser.

SpeechTestMFCC.java is used for running ASR tests with a test set. This class contains methods for running the tests with and without the BSF and finding the WER. It uses cepstrum files as input, which are extracted from a folder containing the wav files to test.

MFCCExtractor.java is used for the extraction of cepstrums from wav files. It is used in both LatticeMFCC.java and SpeechTestMFCC.java.

LatticeDemo.java is used for ASR on a single wav file, without cepstrum extraction.

SpeechTestWav.java is used for running ASR tests with a test set. This class contains methods for running the tests with and without the BSF and finding the WER. Its input is a folder containing the wav files to test.

The classes used for ASR on wav files (LatticeDemo.java and SpeechTestWav.java) will need some work if they are to be used, as they are not up to date with the classes used for ASR on cepstrum files. TaleTUC does not use any source code from these at the moment, and work on them stopped after the data traffic improvement with cepstrum extraction was discovered (see section 7.4.2).

The *Lattice* project also contains three *Sphinx-4* configuration files. These files specify the front-end and other components, and the locations of the *dictionary*, *language model* and *acoustic model*.

config.xml is used in LatticeDemo.java and SpeechTestWav.java, for the decoding of wav files.

config3.xml is used in LatticeMFCC.java and SpeechTestMFCC.java, for the decoding of cepstrum files.

configMFCC.xml is used in MFCCExtractor.java, for the extraction of cepstrums.

Other files in the source package:

corpus.cd_cont_25_wiener_2 contains the latest TaleTUC *acoustic model*, created in *SphinxTrain*. The configration files contain a path to this folder.

Clean contains test files recorded indoors.

Traffic contains test files recorded outdoors.

Mixed contains test files recorded both indoors and outdoors.

MFCCComp contains the cepstrum files converted during runs of the MFCCExtractor.java class.

An illustration of the process of running the LatticeMFCC.java class is shown in figure 10.1, where the numbers indicate the order of the performed operations.

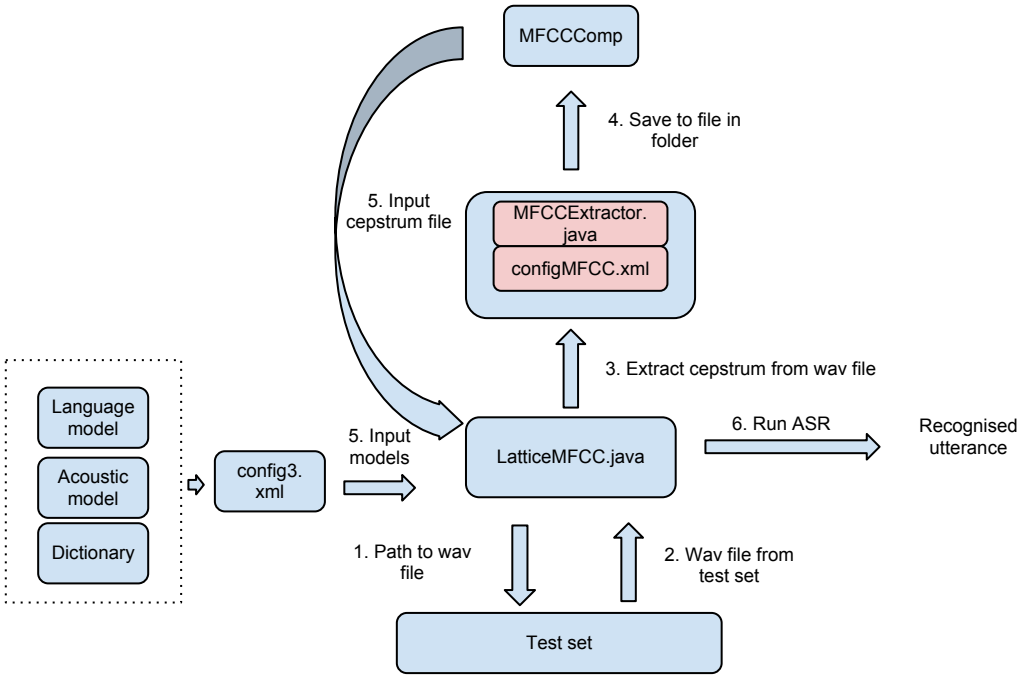

Figure 10.1: LatticeMFCC.java flow chart

The Existing Models

Before one starts to expand the number of speakers or number of bus stop names in TaleTUC's vocabulary, it can be useful to view the latest *language model* and *acoustic model* setup.

The necessary *acoustic model* files are located in */TestTrain/Acoustic_Model/*. This folder contains a folder called *etc*, which includes the setup files, and a folder called *wav*, which contains the audio files. The content of *etc* is listed below.

corpus.dic contains the bus stop names together with phonetic transcriptions.

corpus.lm.DMP contains the *language model*.

corpus.phone contains the phones used in the phonetic transcriptions.

corpus.filler contains the filler phones.

corpus_test.fileids contains the file IDs for all of the test files in the *wav* folder.

corpus_test.transcription contains the transcription of all of the test files in the *wav* folder.

corpus_train.fileids contains the file IDs for all of the training files in the *wav* folder.

corpus_train.transcription contains the transcription of all of the training files in the *wav* folder.

The additional files located in the *etc* folder were generated when the *acoustic model* was built.

To add more speakers to the *acoustic model*, one first has to record the audio. When this is done, the audio files have to be put in the *wav* folder, and the transcription files have to be updated. Which files to update depend on whether the recorded set is to be used for training or testing. The files containing the file IDs also have to be updated to match the number of files in the updated transcriptions.

To add more bus stop names, the first step is to create a new *language model*. TaleTUC's model is located in */TestTrain/Language_Model/*, and is created with the NetBeans project *SphinxTrainTools* (10). The input file needed is called *corpus.txt*, which contains the bus stop names to train the *language model* on. After this list has been updated with the new bus stop names, the *SphinxTrainTools* project can be run. The next step after the *language model* has been built is to add more speakers, as described earlier.

Using the Created Java Program for Training Models

The *language model* and the *acoustic model* have been created in the NetBeans project *SphinxTrainTools*. The relevant classes and files are:

BuildAndTrain.java contains methods for building the *language model*, the *acoustic model* and decoding the *acoustic model*.

Tools.java contains methods for running the *Sphinx* training scripts.

WienerConverter.java contains methods for extracting *Wiener filtered* cepstrums for the training and test data. These files can be used instead of the ones created with *Sphinx-4*'s built-in cepstrum extraction functionality.

config.xml is the config file used by WienerConverter.java.

FileIDAndTranscriptMaker.java is used for creating transcription files based on a text corpus, according to *Sphinx-4*'s rules. To create these transcription files is the first step in creating an *acoustic model* in *Sphinx-4*. They contain

Recording Audio with the Training Application

The training application is an Android Eclipse project that can be mounted through the Eclipse import feature. The audio files recorded with the training application are stored on a server. In the development of TaleTUC, these files were manually copied into the *TestTrain* folder, and organised according to *Sphinx'* rules. The *acoustic model* building Java program (BuildAndTrain.java) maintains a path to this folder, in which it looks for training and test data.

TABuss Git Repository

The TABuss source code can be viewed and pulled from the Git[2] repository at: `https://github.com/saetre/TABuss`. This code also contains the modifications made to TABuss for it to operate as a TaleTUC client. Git repositories can be accessed through the Egit[3], a Git plug-in for the Eclipse IDE[4].

The Server

When the *language model* and the *acoustic model* are ready for use in TaleTUC, they can be uploaded to the server. Files are uploaded to the URL `user@vm-6114.idi.ntnu.no`, with FTP clients such as FileZilla[5], or through SSH file transfers. The *language model* files are located in the */upFiles/Div/etc* folder, and the *acoustic model* is located in the */upFiles/Div/model_parameters* folder. The */upFiles* folder also contains the recordings made with the Android Training application, in addition to the file *bussStop.dic*, which is used by the BSF module.

[2] www.github.com
[3] http://www.eclipse.org/egit/
[4] `http://www.eclipse.org/`
[5] `www.filezilla-project.org/`

The server source code is found in the *SpeechServer* NetBeans project. This project contains the Java Servlets and business logic used. The source code is organised into the following packages:

CBR contains classes for the CBR module.

DTO contains the data transfer objects.

POJO contains the POJOs and Hibernate[6] classes.

core contains the ASR classes and configuration files.

scorer contains the BSF classes for scoring lattice sentences.

servlet contains the Java Servlets.

tools contains the BSF main class, tools and test classes.

To upload a new Web Application Archive (WAR)[7] file, the *SpeechServer* project must first be built. The built WAR file is then be uploaded to the */webapps* folder.

[6] `http://www.hibernate.org/`

[7] `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/WCC3.html`