# CHAPTER 3

# TEXT TO SPEECH SYNTHESIS FOR MOBILE DEVICES – ARCHITECTURE AND DESIGN

## 3.1 INTRODUCTION

The growth of mobile telephony is a global story. The world as a whole is rapidly adopting mobile technology. India's mobile phone industry continues to grow; it will do so by reaching, even more, users who may have difficulty in using a mobile phone with visual interfaces like a keypad or a touch screen.

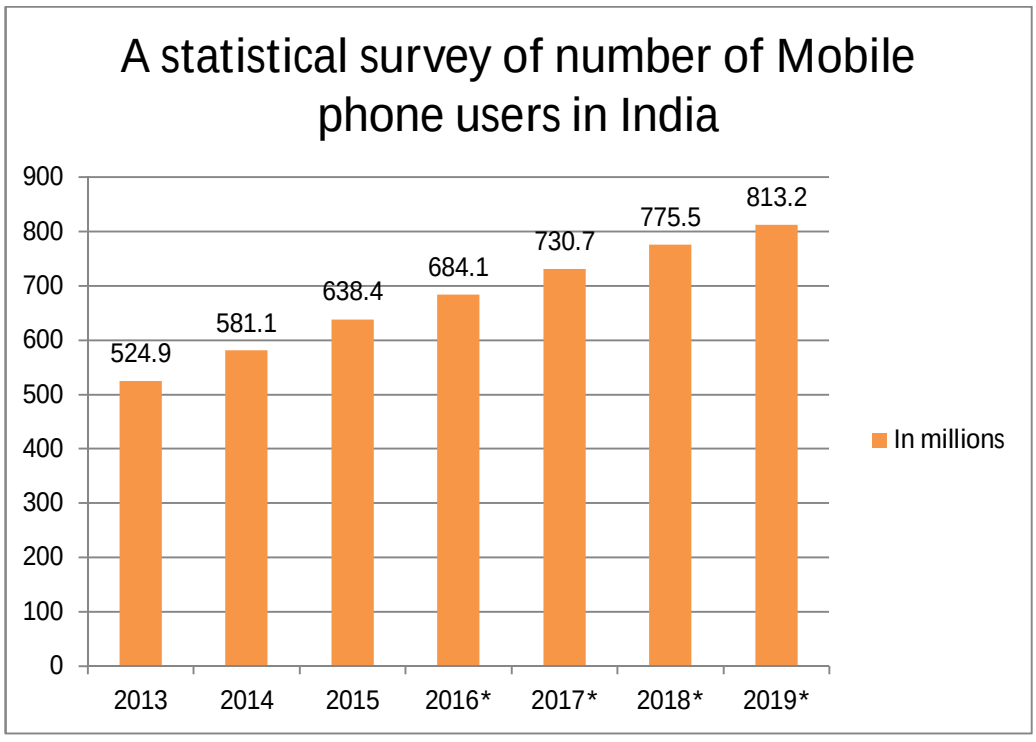


**Figure 3.1 Survey of the number of mobile phone users in India**

India is found to be the third largest smart phone market in the world. The country showcases to have a fastest growing telecom network in the world. Figure 3.1 shows the results of a statistical survey of the number of mobile phone users in India. Statista, one of the leading statistics companies on the internet has published these survey results. With such a proliferation of mobile devices and the emergence of ubiquitous computing, there is now an increasing need for speech interfaces for mobile devices such as cell phones. Many of these mobile devices now have the support for audio interfaces and data entry. Examples of these devices include PDAs, mobile phones, digital diaries, and information points on cars and home appliances. There are voice-activated mobile phones released in Android, Windows, Blackberry, and iOS software. Mobile phone vendors like Samsung, Apple, Microsoft, and Blackberry have put their ideas in marketing voice-activated mobile phones. Text-to-speech synthesis (TTS) is an important component of these speech interfaces. It also finds applications in situations where a visual interface for embedded devices is inappropriate when a visually or physically challenged or an illiterate tries to use such a device.

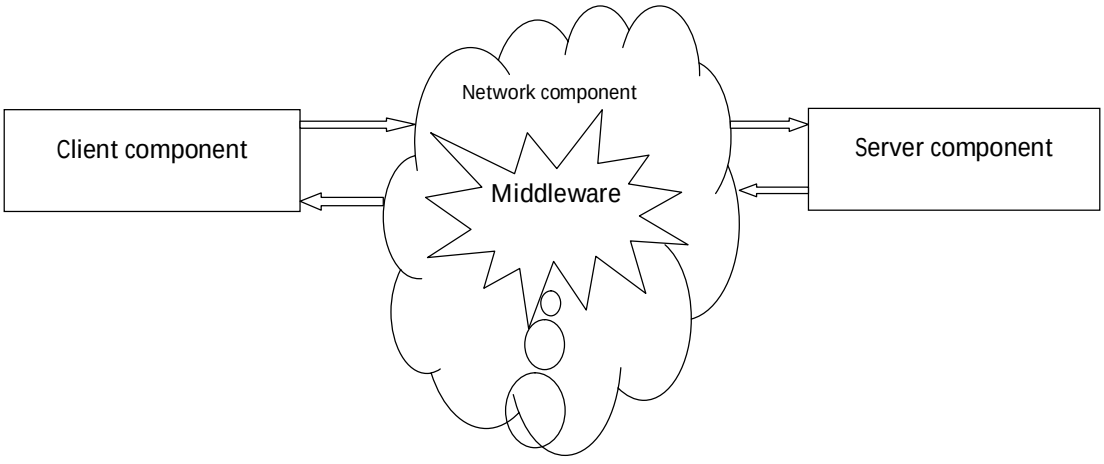


**Figure 3.2 Middleware client-server architecture**

As described in the previous chapters, the extent of the naturalness of synthetic speech produced by state-of-the-art speech synthesizers is mainly

attributed to the use of concatenative synthesis. For an unrestricted speech synthesis, these synthesizers running on high-end machines have RAM and hard disk requirements ranging from 2GB to 4 GB. The memory requirements are high, due to the storage space occupied by the speech repositories and lexicons. These databases hold speech units with different prosodic variations.

In the client-server approach, mobile devices are classified as ‘thin clients’, because of their low computing capabilities and storage space. These embedded devices have chief constraints with their minimal memory resources when compared to machines for which Text-To-Speech synthesizers. Desktops, laptops, and PDAs are classified as normal clients (or ‘fat clients’), as they have their dedicated processors and peripherals. The thin clients are always facilitated with a ‘middleware’, which lies in between the client and the server. Figure 3.2 shows the middleware paradigm. Figure 3.3 shows the role of middleware in a client-server architecture.

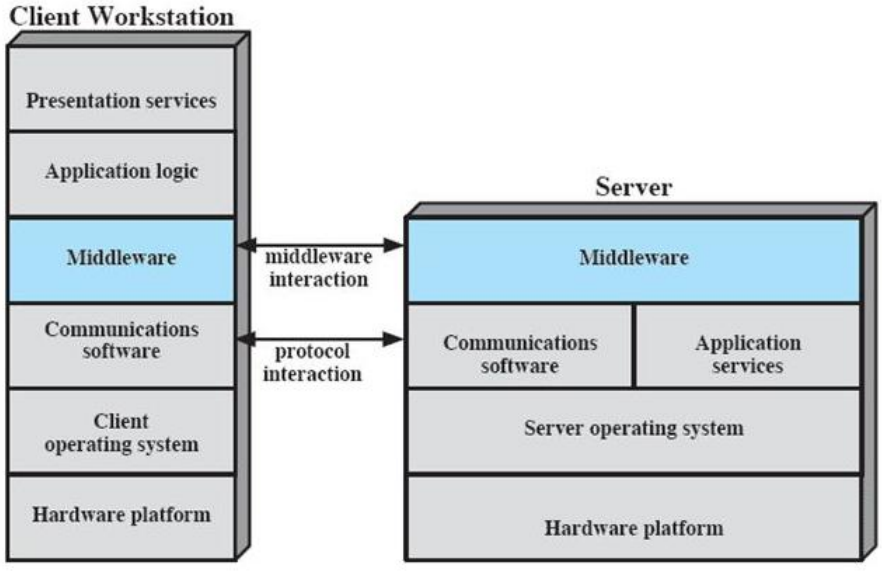


**Figure 3.3 Role of middleware in client-server architecture**

Typically the computing power of an embedded device is between 5 to 50 MIPS (millions of instructions per second). A middleware contributes more for an embedded device, as it bridges the client and the server workstations.

Smart phones allow a part of the resources for vendor-specific programs (they are colloquially called ‘apps’). A Text-To-Speech synthesizer which requires minimal computing power and small memory requirements are a feasible solution. The simplest choice to port these synthesizers for mobile devices by bringing down the repository sizes is a fragile option.

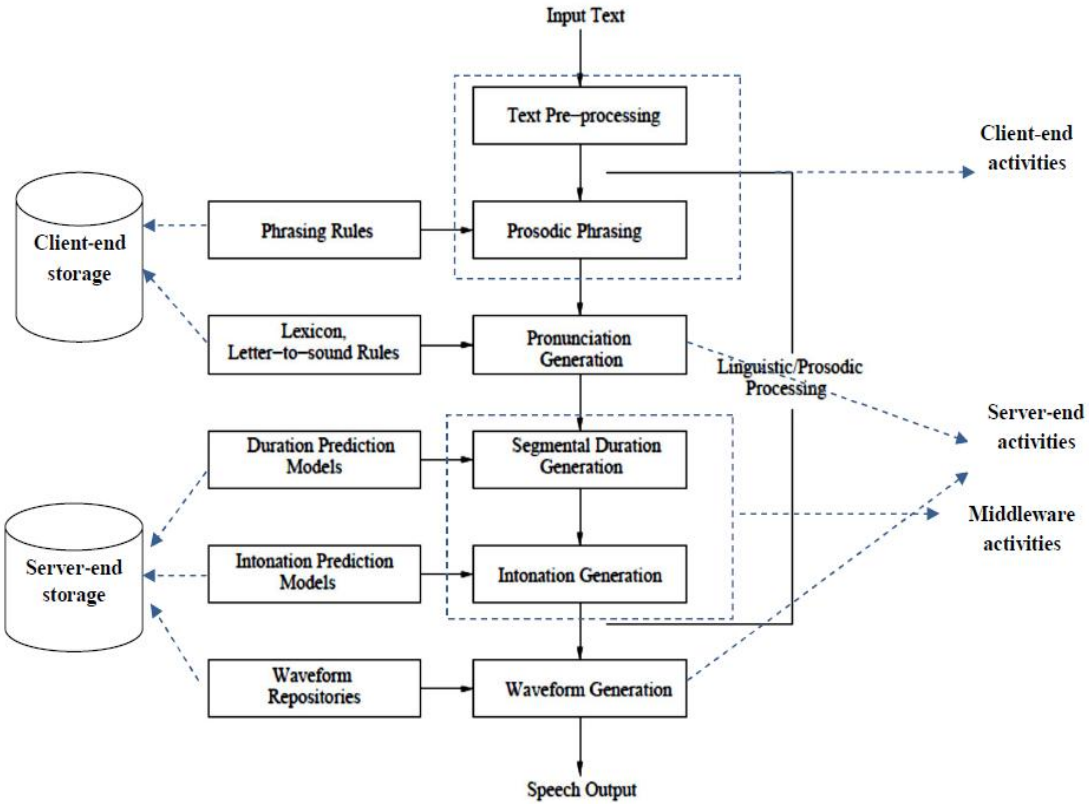


**Figure 3.4 TTS cycle in Distributed Environment**

This trend reduces the number of available speech units and eventually prevents unrestricted speech synthesis. Distributed Speech Synthesis (DSS) shown in the Figure 3.4 is a good alternative, as it allows for

unrestricted speech synthesis on mobile devices with the limited memory and computational capabilities.

Distributed speech synthesis is a client-server approach to speech synthesis. The mobile device acts only as a user-interface for getting the input, pre-processing it, and giving out the speech output. A part of the memory and computational tasks are done at the middleware. The server performs the actual synthesis, thus performing as the perfect back-end. The databases and lexicons are available at the server end. The client-server approach introduces an intermediate component - a network, to transfer intermediate data between the server and the client. DSS systems are built for two speech synthesis systems:

(i) Festival system, the conventional speech synthesis system.

(ii) Flite system, a small and fast, run-time synthesis system used in mobile devices.

Flite is an alternative run-time synthesis platform for Festival. It has a large footprint requiring about 6-10 megabytes of RAM, depending on the language. A user interface is created in the client end, i.e., the mobile device. While grouping the Dravidian languages for speech synthesis, (a recommended extension of our work) the user interface is designed to get any language transliterated input. The UI itself consists of the transliteration options. It collects the Input text in a text box. The front-end of the system (which lies on the client shore – the mobile device)

(i) Receives the transliterated text

(ii) Phrases the text

(iii) Tokenizes the phrased text

(iv) Plays back the output audio file received from the server

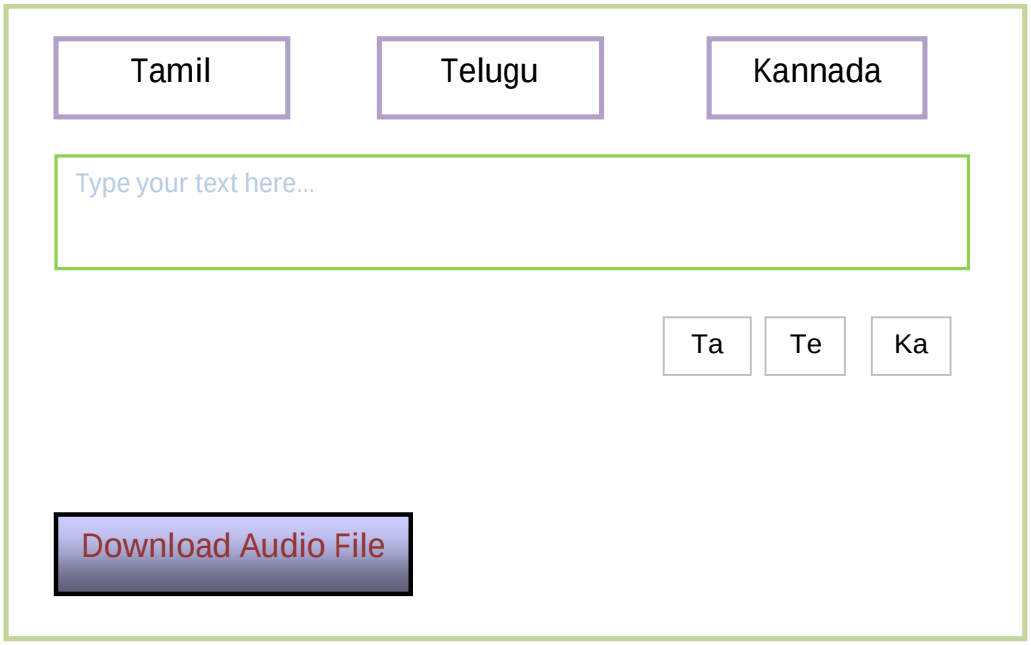


**Figure 3.5 (i) User Interface proposed for the DSS system**

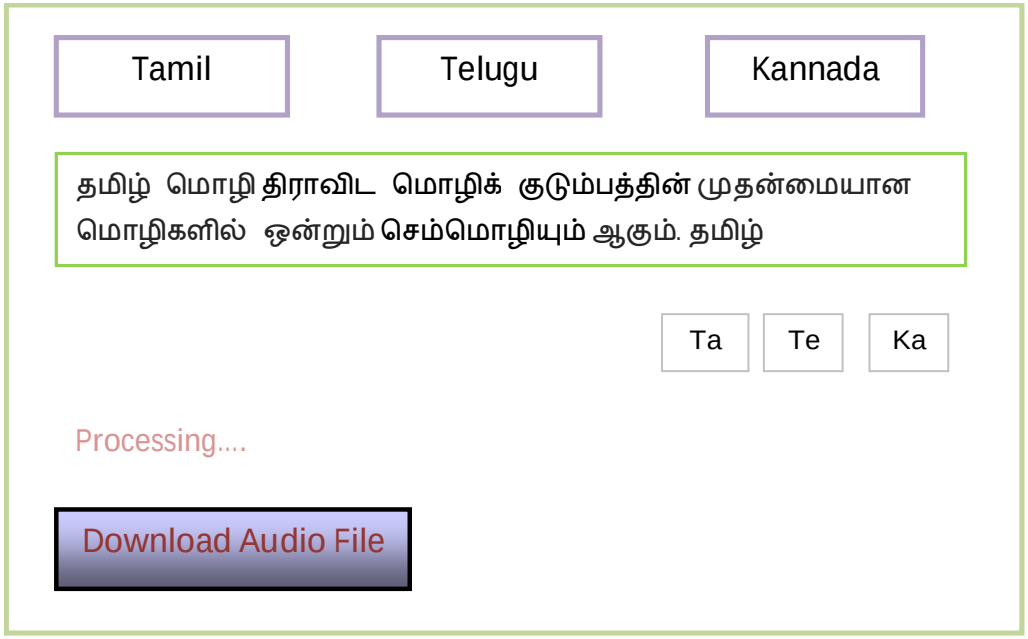


**Figure 3.5 (ii) User Interface on receiving the transliterated Tamil text input**

Figure 3.5 (i) shows the User Interface for DSS system, which runs in the client side. The client end does two different tasks – Reception of language input and plays back the speech output. The input text is the transliterated form of input. The user interface is designed in such a way that multiple languages can be processed, for the convenience of users. Therefore, transliteration options are available for three languages. This design leads to the generalization and scalability of the application. Figure 3.5 (ii) shows the collection of Tamil language input. The client end performs the pre-processing and the server end performs the original synthesis in its complete form. Speech is outputted as waveforms from the server end. It travels through the middleware and reaches the client. The client-end carries out audio file reception and playback. The focus of this chapter is to illustrate methods by which both the synthesizers are run from embedded devices by using a client-server approach to distributing tasks. These methods are suitable for embedded devices as well as standalone systems with low memory and computing capabilities. The next section (Section 3.2) discusses the steps involved in the Festival Speech Synthesis system.

## 3.2 DESIGN OF THE DISTRIBUTED SPEECH SYNTHESIS SYSTEM USING THE CONVENTIONAL FESTIVAL SOFTWARE

As discussed in the previous section, the DSS system primarily has a client and a server. With the DSS introducing a slow network between the client and server, it also becomes necessary to adapt methods to speed up the process by using techniques like multi-threading and buffering.

The first phase accepts text from users for synthesis, performs pre-processing and some basic steps of processing. It sends an intermediate form to the DSS server over the network for further processing. The next phase of the synthesis is carried out on the middleware. The semi-processed output

goes into the server. The server processes the intermediate form, carries the remaining steps of the synthesis and sends back the partly-synthesized speech to the client. After receiving back the intermediate speech data generated by the server, the final phase is to construct a synthetic waveform as the output of synthesis. Figure 3.6 gives the block diagram of this process.

The structure of the Festival software (Black et al. 1998) is given below. Festival uses a data structure called an 'utterance.' An 'utterance' structure starts with text input from the user and ends with a wave file after passing through many phases of synthesis. The various modules of Festival are listed below.

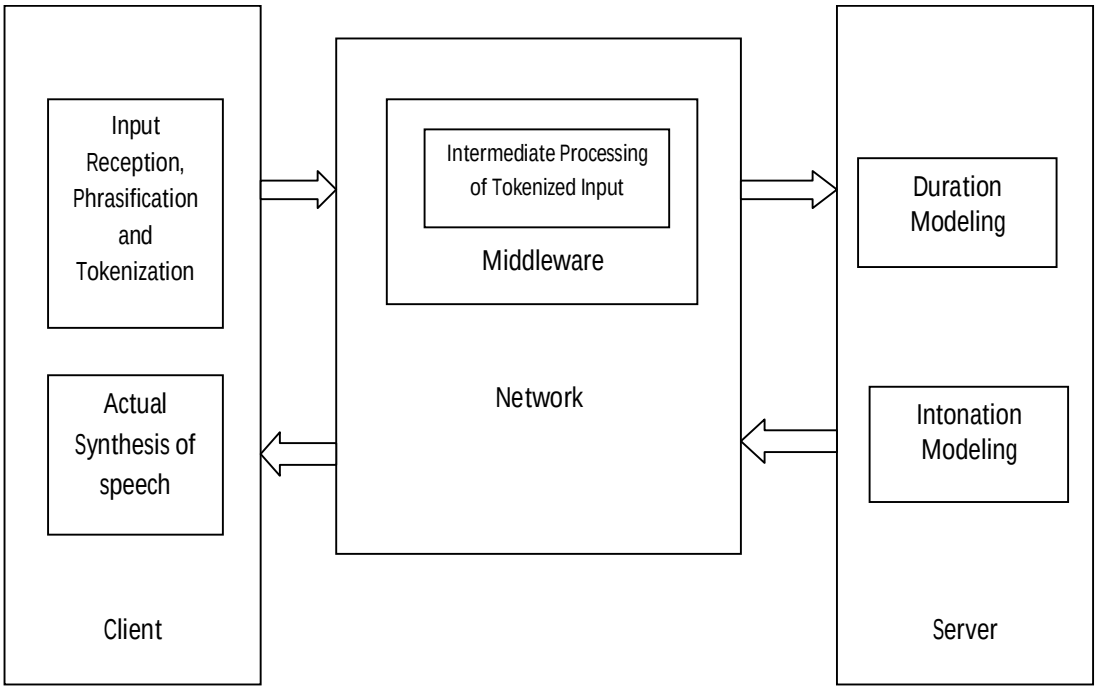


**Figure 3.6 Block Diagram of the DSS system**

(i) The Text Analysis modules (`Token_POS` and `TOKEN`) perform the core token identification. The punctuation marks and irrelevant

white spaces are removed. A given sentence is ready to be treated as a token. It also does homograph disambiguation.

(ii) `POS` module applies part-of-speech rules. It applies the following tags:

a) `pos_lex_name` (Name of the lexicon)

b) `pos_ngram_name` (Name of the n-gram model)

c) `pos_p_start_tag` (At punctuation marks)

d) `pos_pp_start_tag` (At simple nouns)

e) `pos_map` (Name of the actual POS)

(iii) `Phrase_Method` and `Phrasify` modules perform prosodic phrasing. CART trees introduce breaks into the phrases. Statistically trained models assign the following tags to the phrases.

a) `pos_ngram_name`

b) `pos_ngram_filename`

c) `break_ngram_name`

d) `break_ngram_filename`

e) `gram_scale_s` (Weighting factor for breaks)

f) `phrase_type_tree` (CART Tree)

g) `break_tags` (B/NB)

h) `pos_map`

(iv) `Intonation` and `Int_Targets` modules predict accents and boundaries using rules. They call the following sub-modules:

a) Intonation_Default and

b) Intonation_Targets_Default for Default intonations

c) Int_Targets_Simple for simple intonation

(v) `Duration` module predicts the duration of segments using statistically trained models or CART trees. It uses a parameter called Duration_Method. The various values set for this parameter are:

a) `Duration_Stretch` which assigns a particular value for duration

b) `Default`, for Default duration

c) `Averages`, for Average duration

d) `Klatt`, for duration values from the `Klatt` book

e) `Duration_cart_tree`, for Duration, obtained using CART tree.

(vi) `Int Targets` module generates F0 values for each identified speech unit

(vii) `UniSyn` module generates the synthesized waveform for the text input using:

a) 'Pitchmark' program

b) 'Sig2fv' command

Of these, modules till the POS stage do not require any large databases or involve complex operations (Samuel Thomas, 2007). The DSS system is visualized as follows:

a) DSS client. It performs the Token_POS and TOKEN phases and writes an ‘utterance’ structure to disk, which is then transferred over the network.

b) DSS server which loads the ‘utterance’ structure into the framework.

Instead of using the default speech synthesis functions, the server now applies the remaining speech synthesis functions on the ‘utterance’ structure one after another till completion.

A straight forward implementation of the client and server as described above. After the entire wave file is received at the client, the playback to the user begins at the client end. However, this simple implementation results in a considerably large delay between the time the user enters text at the client end and the time the speech output is available. This is because of data moving twice across the intermediate network. It is, therefore, essential to design the client and server for the DSS with suitable techniques to reduce this undesired delay.

## 3.3 DESIGN OF THE DISTRIBUTED SPEECH SYNTHESIS SYSTEM USING FLITE SOFTWARE

The Flite distribution consists of two distinct parts (Black et al. 1998):

a) The Flite library, which contains the core synthesis code

b) Voice(s) for Flite. A voice library that contains three sub-parts

(i) Language models such as text processing, prosody models, etc.

(ii) Lexicons and letter to sound rules

(iii) Unit database and voice definition

The following functions are used for speech synthesis in the Flite system:

(i) `flite_init` – Initializes the procedures.

(ii) `flite_text_to_wave` – Returns a waveform of a given string of text.

(iii) `file_to_speech` – Gets a filename as the input and returns the synthesized speech form of the file.

(iv) `flite_text_to speech` – Gets the string of text as the input. The programmer can select a particular voice from the voice library. The synthesized speech gets stored in the insisted filename. This function returns the size of the speech file in seconds.

(v) `flite_synth_text` - Gets the string of text as the input and synthesizes the text into speech with the selected voice. It returns the utterance from it, for further processing.

(vi) `flite_synth_phones` - Gets the series of phones as the input and synthesizes the text into speech with the selected voice. It returns the utterance from it, for further processing.

(vii) `flite_voice_select` – Helps the programmer to identify and select the desired voice. The function returns a pointer to the voice with the specified name.

(viii) `flite_ssml_file_to_speech` - Gets input a filename in the form of SSML (Speech Synthesis Markup Language). Returns the synthesized speech form of the file.

(ix) `flite_ssml_text_to_speech` - Gets input a SSML text and returns the synthesized speech form of the file.

(x) `flite_voice_add_lex_addenda` - loads the pronunciations from the specified file into the lexicon identified in the given voice. This process will cause all other voices using that lexicon also to get this new addenda list.

The function flite_voice_add_lex_addenda creates the lexicon and letter-to-sound rule. The functions flite_text_to_wave, file_to_speech and flite_text_to speech are used for verifications and validations of the synthesized speech units during the unit-creation phase. The function flite_voice_select is used to select the appropriate voice for synthesizing the speech. The functions flite_synth_text and flite_synth_phones are used for creating utterances after proper validation of synthesized speech. The functions flite_ssml_file_to_speech and flite_ssml_text_to_speech are used for train data.

## 3.4 CONCLUSIONS

The designs for the DSS system are presently implemented and tested on desktop machines. Apart from the DSS server, the client and mobile network are only simulations.

The DSS server is implemented in C++ and uses the Festival API set for C/C++ to interface to the Festival synthesis framework. The DSS client is built with Symbian C++ using the Series 60 2.1 Platform Software Development Kit (SDK) from Nokia for the Symbian OS 7.0s. The Platform SDK has a rich set of APIs for implementing the client designs discussed in the previous section. The implementation runs using the emulator available with the Platform SDK (Flite Software Manual).

The performance of the DSS system is assessed using the Mean Opinion Scores on a 5-point scale, with 10 native Tamil speakers. They are asked to score the naturalness of each output on a scale from 1 to 5 (1=Bad, 2=Poor, 3=Fair, 4=Good 5=Excellent). The two paragraphs of the text had a total number of 24 sentences with 10 to 70 characters per sentence. It is seen that the time taken by the distributed synthesis system is almost equal to that of the standalone systems. The mean opinion scores are somewhat low primarily because of the prominent gaps that appear in the synthesis of long sentences. The acceptability rate may be increased by dynamically dividing long sentences at suitable points. Increasing the network bandwidth can also reduce the time required for synthesis. Issues related to actual implementation are also addressed in this thesis. These results could also lead to the development of an ASIC for text-to-speech systems for embedded systems. The next chapter describes the Festival speech system architecture and the automatic generation of speech units.