BUILDING TOOLS USING HINDI SPEECH RECOGNIZER

SACHIN JOSHI
FOSS.IN
25-29 Nov 2008

Contents

- A brief intro to Automatic Speech Recognition (ASR)
  - What is speech recognition ?
  - Types of ASR
- What are the components of ASR ?
- Open Source Tools for Speech Recognition
- Overview of Hindi ASR
- Application areas
- Tightly coupled Vs loosely coupled applications
- Three things an application developer should know
  - Phone set
  - Creating Phonetic Lexicon
  - Creating domain specific Language Model
  - Using CMU Sphinx APIs
- Sphinx 2 API overview
- Anatomy of live decoder
- Demo dialog system
- Architecture of railway dialog system

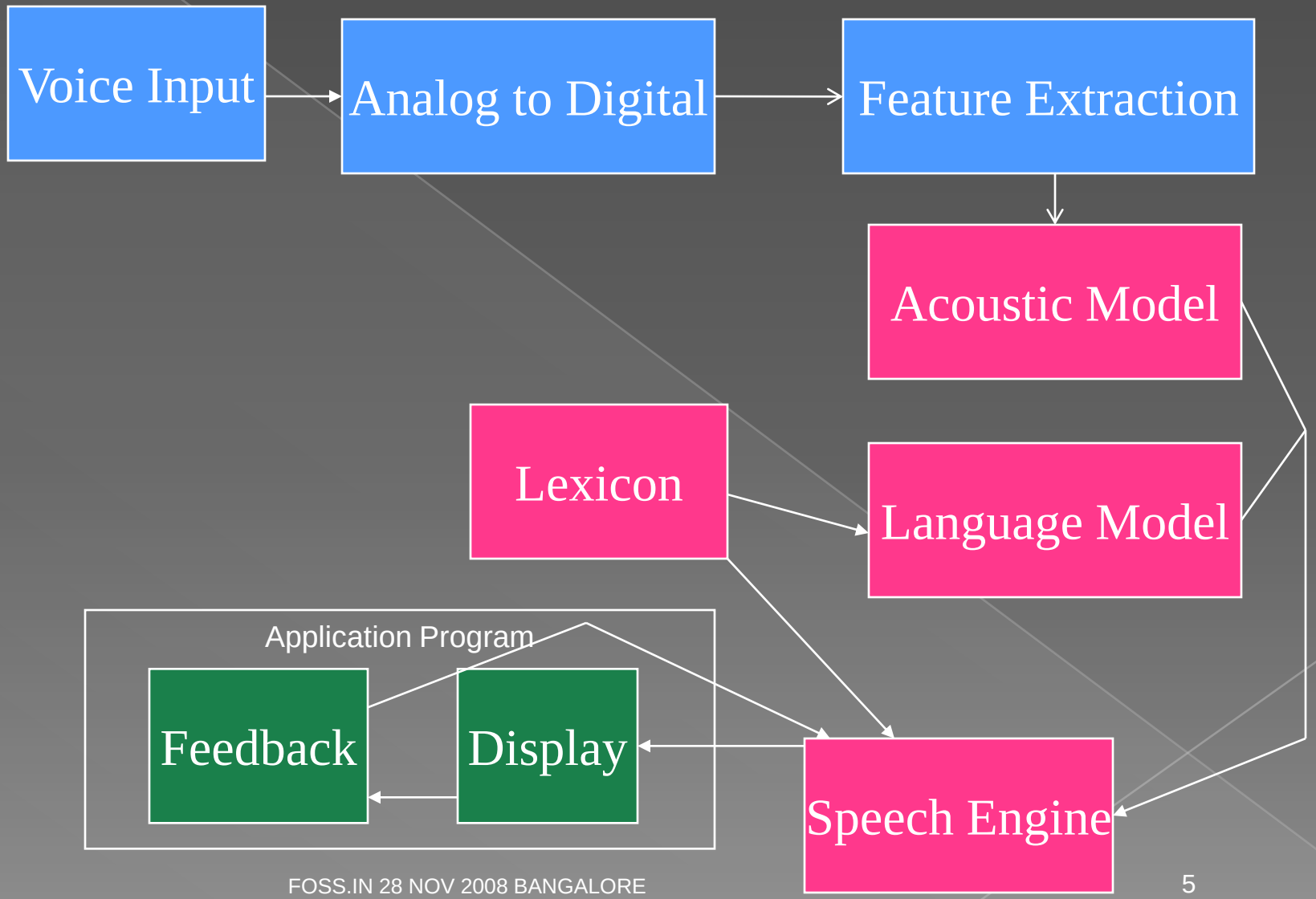
A Brief Intro To Automatic Speech Recognizer (ASR)

- What is speech recognition
  - Is it voice recognition ? No.
  - Recognizing the sequence of predefined set of units (like phonemes) spoken by a speaker.

A Brief Intro To Automatic Speech Recognizer (ASR)

- Types of Speech Recognition Systems
  - Based on Technology
    - Continuous Vs Isolated Word Recognizer
  - Based on application
    - Limited Domain Vs Unrestricted
  - Based on Decoder Mode
    - Live Mode Vs Batch Mode
  - Based on Number of Users
    - Single Speaker Vs Multispeaker

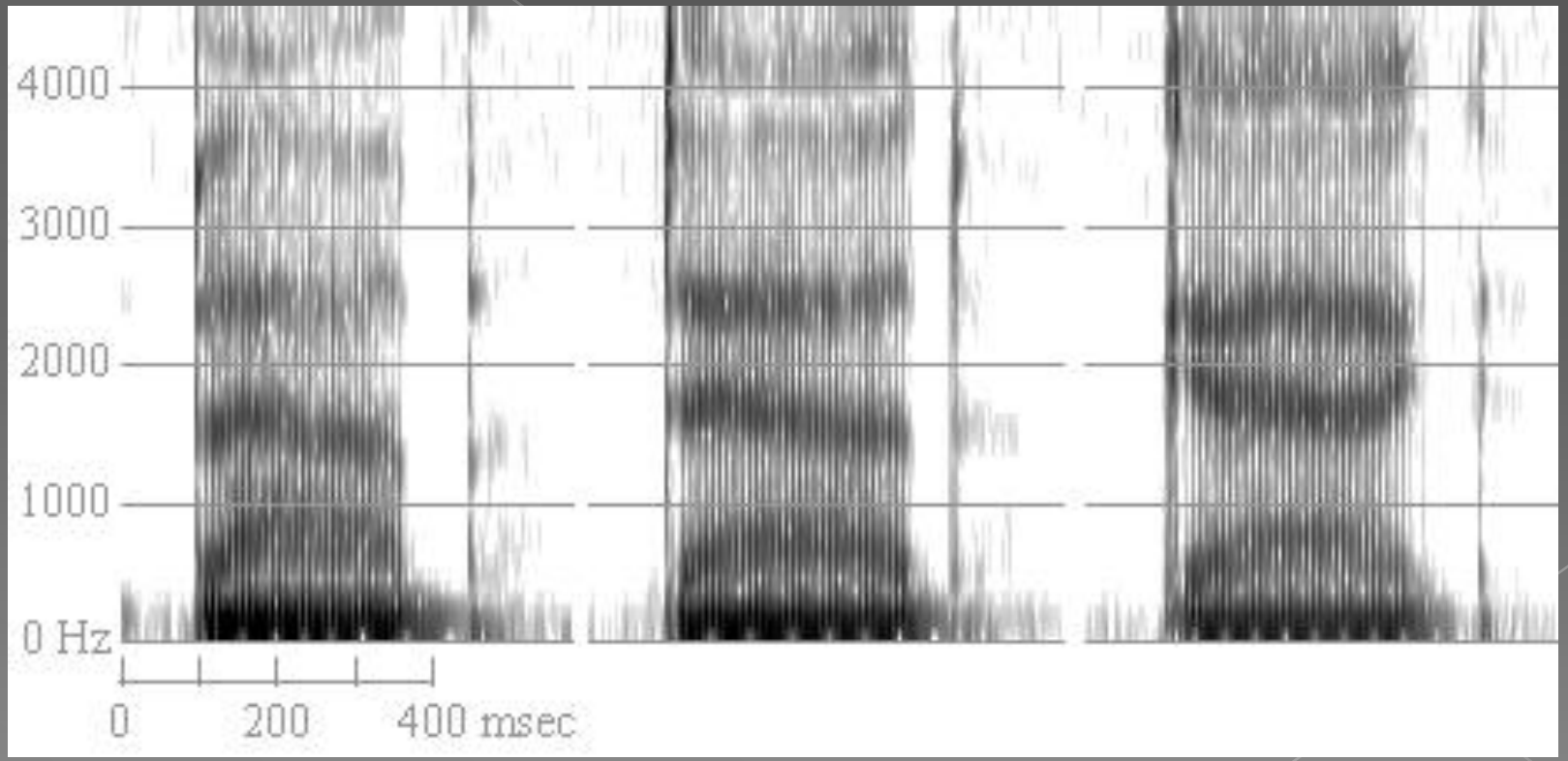
Components of ASR



Components of ASR

- Acoustic model
  - Captures the spectral variations of every phoneme

bab dad gag



Courtesy : http://home.cc.umanitoba.ca/~robh/howto.html

Components of ASR

- Acoustic model
  - Uses technique called as Hidden Markov Models to capture the temporal variations in speech data
  - HMM is a statistical Machine Learning Technique
  - All the models are stored in form of binary files which are then used by decoder for comparing with actual speech data

Components of ASR

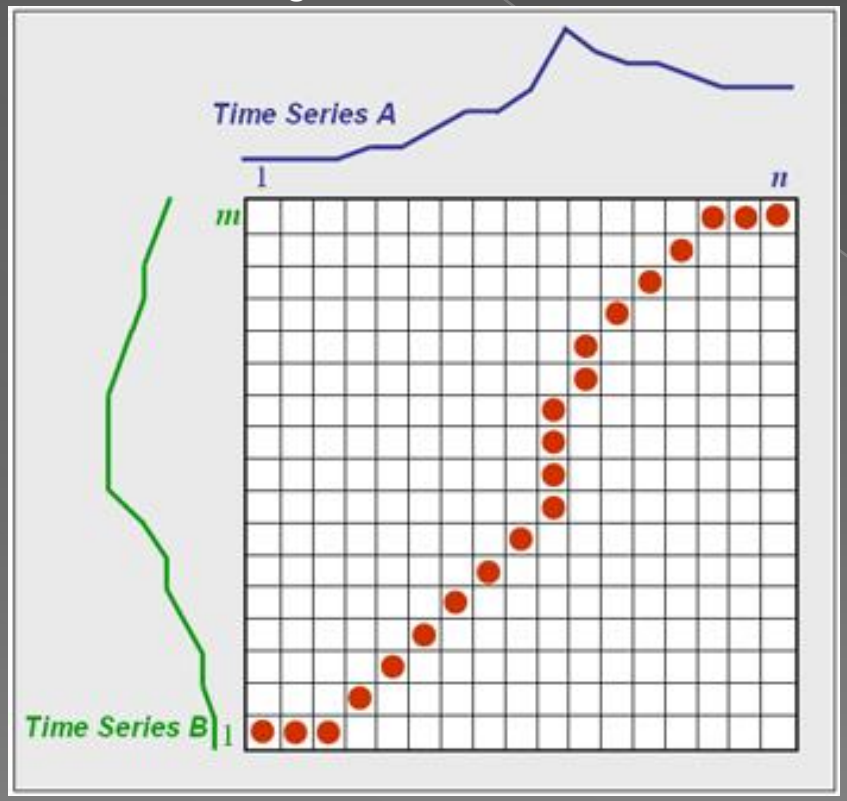
- Phonetic Lexicon
  - Gives phonetic representation of every word in vocabulary
  - Can assemble valid words from output produced by acoustic model
    - चेयरमैन च् ए र् अ म् ऐ न्
    - आशावादी आ श् आ व् आ द् ई
  - Mantra of ASR based applications - Smaller the vocabulary size better is the recognition accuracy

Components of ASR

- Language Model
  - Captures the underlying grammatical structure of language in statistical framework
  - Bigrams –
    - Capture probability of one word occurring after another
      - आशावादी चेयरमैन - quite probable bigram
  - Trigrams –
    - Capture probability of one word occurring a bigram
      - आशावादी चेयरमैन चेयरमैन – highly rare trigram
      - आशावादी चेयरमैन ने – possible trigram
  - Language model plays very important role in speech applications
  - Good language model improves performance of ASR by large extent

Components of ASR

DTW matching of two time series



Courtesy: http://www.tu-plovdiv.bg/Container/bi/DTWimpute/DTWalgorithm.html

- Speech Engine / Decoder
  - Compares input speech data with acoustic models
  - Uses modified version of basic DTW algorithm
  - Uses Hidden Markov Model (HMM) based acoustic phonetic models for comparison

Components of ASR

- Analog to Digital Convertor
  - Done by sound card
- Noise Filtering
  - Done by few well known available algorithms
  - Mostly done by technique known as pre-emphasizing
- Feature Extraction
  - Features represent compact representation of information contained in signal
  - They reduce computational load
  - Most well known feature Mel-frequency cepstral coefficients (MFCC) used in Hindi ASR

Open Source Tools for Speech Recognition

- Creating Acoustic Models
  - HTK, CMU SphinxTrain
- Creating Phonetic Lexicon
  - For English reference dictionaries are available
  - For other languages in-house code
- Language Modeling
  - CMUCLMTK, SimpleLM, SRI Labs, HTK
- Speech Engine / Decoder
  - Hdecode (HTK), CMU Sphinx

Open Source Tools for Speech Recognition

- HTK can be used only for research purpose
- Sphinx is also developed as a research tool but it is freely available for system building also
  - Sphinx 2 – uses semicontinuous models
  - Sphinx 3 – uses continuous models written in C
  - Sphinx 4 – Java implementation of Sphinx 3
  - Pocket Sphinx – A decoder for PDA, mobile devices
  - http://cmusphinx.sourceforge.net/html/cmusphinx.php
- Julius is only speech engine which can use models built with HTK trainer

Overview of Hindi ASR

- Features
  - A multi-speaker large vocabulary continuous speech recognition system
  - Developed using CMU Sphinx Tools
  - Applicable for native north Indian Hindi speakers particularly from Delhi, Madhya Pradesh, Uttar pradesh, some part of Rajasthan and Punjab
  - Applicable for (but not necessarily restricted) to urban speakers in particular
  - Applicable for age group from 18 to 60 years
  - Applicable for both male and female speakers
  - Can be plugged into any application program using CMU Sphinx APIs
  - Models built on 16KHz data recorded on Desktops and Laptops

Overview of Hindi ASR

- Hindi ASR was developed under Sarai FLOSS Fellowship 2007
- Objective:
  - To build generic acoustic models for multi-speaker Hindi speech recognizer to enable open source community to develop spoken user interfaces in Hindi language
- 40 speakers in total
- 50 sentences per speaker
- Recordings on phonetically balanced corpus
- Microphone and computer variations covered

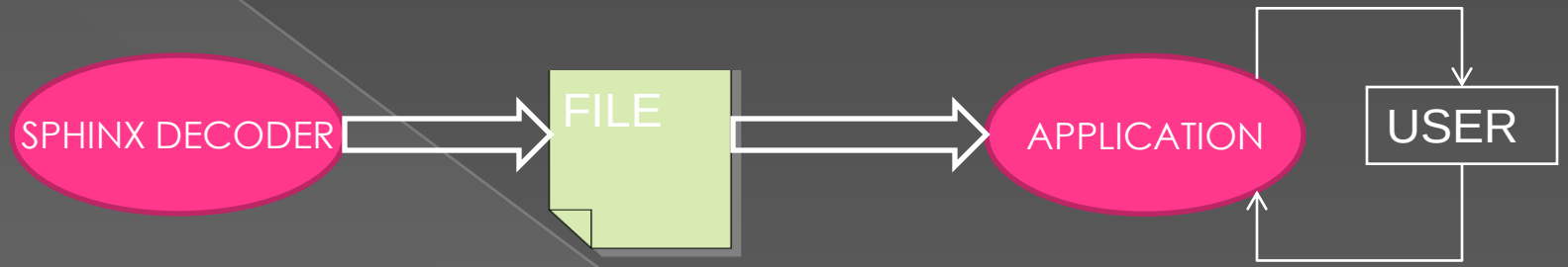
Overview of Hindi ASR

http://sourceforge.net/projects/hindiasr

Application Areas

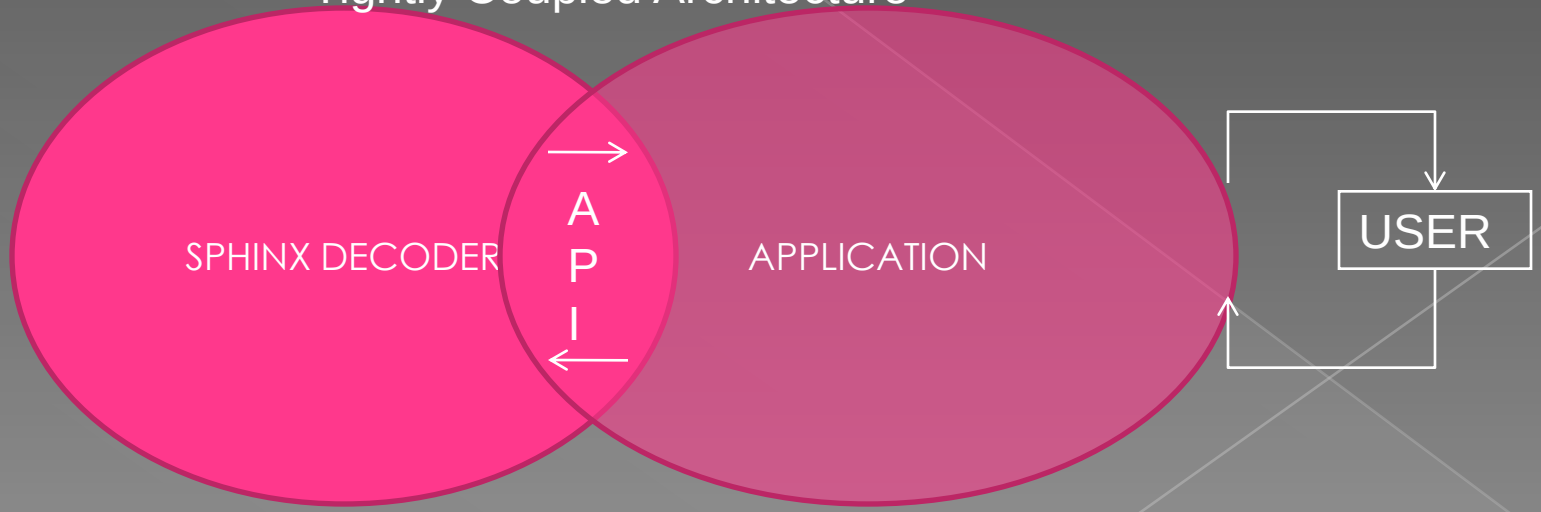
- Dictation
  - For specific domains like office documents
- System control/navigation
  - For driving menus, pushbuttons and GUI components
  - Building hands free GUI interfaces
- Commercial/Industrial applications
  - Limited domain intelligent dialog systems
  - Telephonic inquiry systems (IVR)
- Software for blind people
- Entertainment
  - Operations in computer games
  - Building chatter bots
- Driving intelligent devices
  - Robots, washing machines, coffee vending machines

Loosely Vs Tightly Coupled Applications



Loosely Coupled Architecture

Tightly Coupled Architecture



Loosely Vs Tightly Coupled Applications

- Tightly coupled Application Architecture
  - The application linked to Sphinx libraries
  - Application can access sphinx APIs
  - Application has better control over many things in speech engine
    - Speech signal acquisition
    - Buffering
    - Feature extraction
    - Decoder configurations
    - Decoder process and outputs
    - Acoustic and Language model scores
    - Changing Language model
  - All advanced applications like user interfaces should be ideally implemented in tightly coupled architecture

Loosely Vs Tightly Coupled Applications

- Loosely coupled Application Architecture
  - The application uses Sphinx decoder as a black box
  - Application can not access sphinx APIs
  - The coupling between application and Sphinx recognition engine is by writing and reading the files
  - The only information which Application gets from Sphinx is decoded string and final scores for that in some predefined format
  - This kind of architectures slows down performance of whole application since Sphinx runs as a separate process and your application actually waits for decoder output
  - Advantage :
    - Simple and quicker to implement
    - No knowledge of sphinx APIs is needed

Three things an application developer should know

- Phone set
- Creating Phonetic Lexicon (dictionary)
- Creating Domain Specific Language Model
- How to use Sphinx APIs

Hindi ASR Phone set

- Phoneme is a basic unit of sound in a particular language
- Transliteration scheme called Itrans-3 is used to represent basic phonemes
- a aa r I k k~ n ei s t oo l p ii n: y m v u d g g~ b h j t: ch sh d: uu dh shh nd~ bh ai au th ph ph~ kh kh~ d~ rx chh t:h gh d:h jh dh~ o o- nj~ ng~ l: h: e- e j~ SIL
- Number of phonemes - 59

Creating Phonetic Lexicon

- A phonetic lexicon contains words and their phonetic contents

  E.g. kahaan:vat k a h aa n: v a t

  lad:akaa l a d: k aa

- Phonetizer
  - A program which converts a give word into phonemes
- Issues
  - The text is always in Unicode, iscii like encodings
  - We need another good convertors which can convert these formats into Itrans-3 notations

Creating Phonetic Lexicon

- Decide Application Vocabulary
  - Guess all the words which user is likely to say
  - Cover all possible words
- Phonetize all these words and create phonetic lexicon.
- Some words may have more than one phonetic representation e.g.

  tiin t ii n

  Tin t i n

Creating Phonetic Lexicon

- ATTENTION !
  - Phonetic lexicon must contain all the words which you are intending to recognize in your application
  - The decoder can not recognize any word which is outside this lexicon !

Creating Domain Specific Language Model

- Create a file containing the possible verbal commands, sentences and phrases which can be spoken out by user for that particular application
- Try to cover all possibilities
- CMU Cambridge Language Modeling toolkit (cmuclktk) will be used to build a language model

Creating Domain Specific Language Model

- Steps in creating language model
  - Create word frequencies
  - Create vocabulary file
  - From corpus, word frequencies and vocabulary file, create N-gram file
  - Finally create language model file which is in arpa format

Sphinx 2 API overview

- APIs devided into three classes
  - Raw audio access
  - Continuous listening and silence filtering
  - Core decoder functionality

Sphinx 2 API overview

- Core decoder never accesses audio devices itself
- Application has to take care of reading audio data and sending it to decoder
- This gives freedom and choice to application, to decode any audio data randomly

Raw Audio Access

- Audio device interface changes from platform to platform
- Sphinx provides APIs encapsulating device specific code. It provides a common interface to application developer

Raw Audio Access APIs

- **For recording**
  - **ad_open:** Opens an audio device for recording. Returns a *handle* to the opened device. (Currently 8KHz or 16KHz mono, 16-bit PCM only.)
  - **ad_start_rec:** Starts recording on the audio device associated with the specified handle.
  - **ad_read:** Reads up to a specified number of samples into a given buffer. It returns the number of samples actually read, which may be less than the number requested. In particular it may return 0 samples if no data is available. Most operating systems have a limited amount of internal buffering (at most a few seconds) for audio devices. Hence, this function must be called frequently enough to avoid buffer overflow.

Raw Audio Access APIs

- **ad_stop_rec:** Stops recording. (However, the system may still have internally buffered data remaining to be read.)
- **ad_close:** Closes the audio device associated with the specified audio handle.

- **For playback**
  - **ad_open_play:** Opens an audio device for playback. Returns a *handle* to the opened device. (Currently 8KHz or 16KHz mono, 16-bit PCM only.)
  - **ad_start_play:** Starts playback on the device associated with the given handle.

Raw Audio Access APIs

- **ad_write:** Sends a buffer of samples for playback. The function may accept fewer than the samples provided, depending on available internal buffers. It returns the number of samples actually accepted. The application must provide data sufficiently rapidly to avoid breaks in playback
- **ad_stop_play:** End of playback. Playback is continued until all buffered data has been consumed.
- **ad_close_play:** Closes the audio device associated with the specified handle.
- **ad_mu2li** for converting 8-bit mu-law samples into 16-bit linear PCM samples.

Continuous Listening and Silence Filtering

- Live Decoder need to continuously listen to input signal and determine which part of signal is speech and filter out silence durations
- Sphinx 2 can only decode utterances that are limited to less than about 1 min so application has to pass the data to decoder in chunks and get it decoded

Continuous Listening and Silence Filtering APIs

- **cont_ad_init:** Associates a new continuous listening module instance with a specified raw A/D *handle* and a corresponding *read* function pointer. E.g., these may be the handle returned by **ad_open** and function **ad_read** described above.

- **cont_ad_calib:** Calibrates the background silence level by reading the raw audio for a few seconds. It should be done once immediately after **cont_ad_init**, and after any environmental change.

- **cont_ad_read:** Reads and returns the next available block of non-silence data in a given buffer. (Uses the *read* function and *handle* supplied to **cont_ad_init** to obtain the raw A/D data.) More details are provided below.

- **cont_ad_reset:** Flushes any data buffered inside the module. Useful for discarding accumulated, but unprocessed speech.

Continuous Listening and Silence Filtering APIs

- **cont_ad_get_params:** Returns the current values of a number of parameters that determine the functioning of the silence/speech detection module.
- **cont_ad_set_params:** Sets a number of parameters that determine the functioning of the silence/speech detection module. Useful for fine-tuning its performance.
- **cont_ad_set_thresh:** Useful for adjusting the silence and speech thresholds. (It's preferable to use **cont_ad_set_params** for this purpose.)
- **cont_ad_detach:** Detaches the specified continuous listening module from its currently associated audio device.
- **cont_ad_attach:** Attaches the specified continuous listening module to the specified audio device. (Similar to **cont_ad_init**, but without the need to calibrate the audio device. The existing parameter values are used instead of being reset to default values.)

Continuous Listening and Silence Filtering APIs

- **cont_ad_close:** Closes the continuous listening module.

- In addition to returning non-silence data, the function **cont_ad_read** also updates a couple of parameters that may be of interest to the application:
  - The *signal level* for the most recently read data.
  - A *timestamp* value which is useful to determine the length of silence between two speech utterances

Speech-to-Text Decoding APIs

- There are several aspects of speech decoding
  - Initialization of decoder configuration
  - Basic speech decoding
  - Managing multiple Language Models

- Initialization of decoder configuration

  - **fbs_init:** Initializes the decoder. In case of live mode decoding the option –ctlfn should not be used

  - **fbs_end:** Cleans up the internals of the decoder, such as printing summaries and closing log files, before the application exits.

Speech-to-Text Decoding APIs

- Actual Decoding
  - **uttproc_begin_utt:** Begins decoding the next utterance. The application can assign an *id* string to it. If not, one is automatically created and assigned.
  - **uttproc_rawdata:** Processes (decodes) the next chunk of raw A/D data in the current utterance. This can be *non-blocking*, in which case much of the data may be simply queued internally for later processing. Note that only single-channel (mono) 16-bit linear PCM-encoded samples can be processed.
  - **uttproc_cepdata:** This is an alternative to **uttproc_rawdata** if the application wishes to decode *cepstrum* data instead of raw A/D data.
  - **uttproc_end_utt:** Indicates that all the speech data for the current utterance has been provided to the decoder.

Speech-to-Text Decoding APIs

- **uttproc_result:** Finishes processing internally queued up data and returns the final recognition result string. It can also be *non-blocking*, in which case it may return after processing only some of the internally queued up data.

- **uttproc_result_seg:** Like **uttproc_result**, but returns additional information for each word in the result, such as time segmentation (measured in 10msec frames), acoustic and language model scores, etc. (See structure **search_hyp_t** in file **include/fbs.h**.) One can use either this function or **uttproc_result** to finish decoding, but not both.

- **uttproc_partial_result:** This function can be used to obtain the most up-to-date partial result while utterance decoding is in progress. This may be useful, for example, in providing feedback to the user.

- **uttproc_partial_result_seg:** Like **uttproc_partial_result**, but returns word segmentation information (measured in 10msec frames) instead of the recognition string.

Speech-to-Text Decoding APIs

- **uttproc_abort_utt:** This is an alternative to **uttproc_end_utt** that terminates the current utterance. No further recognition results can be obtained for it.
- **search_get_alt:** Returns N-best hypotheses for the utterance. Currently, this does not work with finite state grammars. (See further details in **include/fbs.h**)

- The *non-blocking* option in some of the above functions is useful if decoding is slower than real-time, and there is a chance of losing input A/D data if processing them takes too long.
- The code fragment for actual decoding step actually looks like following –

```
uttproc_begin_utt (....)
while (not end of utterance) { /* indicated externally, somehow */
        read any available A/D data; /* possibly 0 length */
        uttproc_rawdata (A/D data read above, non-blocking);
}
 uttproc_end_utt ();
uttproc_result (...., blocking);
```

Speech-to-Text Decoding APIs

- Managing multiple LMs
  - **lm_read:** Reads in a new N-gram language model from a given file, and associates it with a given string name. The application needs this function only if it needs to create and load LMs dynamically at run time, rather than at initialization via the **-lmfn** command line argument.
  - **lm_delete:** Deletes the N-gram LM with the given string name from the decoder repertory.
  - **uttproc_set_lm:** Tells the decoder to switch the active grammar to the N-gram LM with the given string name. Subsequent utterances are decoded with this grammar, until the next
  - **uttproc_set_lm** or **uttproc_set_fsg** operation. This function can only be invoked between utterances, not in the midst of one.

Speech-to-Text Decoding APIs

- **uttproc_set_context:** Sets a two-word history for the next utterance to be decoded, giving its first words additional context that can be exploited by the LM. (Useful only with N-gram LMs.)
- **uttproc_load_fsgfile:** Loads the given finite-state grammar (FSG) file into the system and returns the string name associated with the FSG. (Unlike the N-gram LM, the string name is contained in the FSG file.) The application needs this function only if it needs to create and load FSGs dynamically at run time, rather than at initialization via the **-fsgfn** or **-fsgctlfn** command line arguments.
- **uttproc_load_fsg:** Similar to **uttproc_load_fsgfile**, but the input FSG is provided in the form of an **s2_fsg_t** data structure (see **include/fbs.h**), instead of a file.
- **uttproc_set_fsg:** Tells the decoder to switch the active grammar to the FSG with the given string name. Subsequent utterances are decoded with this grammar, until the next **uttproc_set_fsg** or **uttproc_set_lm** operation. This function can only be invoked between utterances, not in the midst of one.
- **uttproc_del_fsg:** Deletes the FSG with the given string name from the decoder repertory.

Speech-to-Text Decoding APIs

- Data logging
  - The raw input data for each utterance and/or the cepstrum data derived from it can be logged to specified directories:
  - **uttproc_set_rawlogdir:** Specifies the directory to which utterance audio data should be logged. An utterance is logged to file <id>.raw, where <id> is the string ID assigned to utterance by **uttproc_begin_utt**.
  - **uttproc_set_mfclogdir:** Specifies the directory to which utterance cepstrum data should be logged. Like A/D files above, an utterance is logged to file <id>.mfc.
  - **uttproc_get_uttid:** Retrieves the utterance ID string for the current or most recent utterance. Useful for locating the logged A/D data and cepstrum files, for example.

Anatomy of Live Decoder

```
/*
 * Main utterance processing loop:
 *     for (;;) {
 *         wait for start of next utterance;
 *         decode utterance until silence of at least 1 sec observed;
 *         print utterance result;
 *     }
 */
static void utterance_loop()
{
    int16 adbuf[4096];
    int32 k, fr, ts, rem;
    char *hyp;
    cont_ad_t *cont;
    char word[256];

    /* Initialize continuous listening module */
    if ((cont = cont_ad_init (ad, ad_read)) == NULL)
        E_FATAL("cont_ad_init failed\n");
    if (ad_start_rec (ad) < 0)
        E_FATAL("ad_start_rec failed\n");
```

Anatomy of Live Decoder

```
/* Calibration */
if (cont_ad_calib (cont) < 0)
        E_FATAL("cont_ad_calib failed\n");

    for (;;) {
        /* Indicate listening for next utterance */
       printf ("READY....\n"); fflush (stdout); fflush (stderr);
        /* Await data for next utterance */
        while ((k = cont_ad_read (cont, adbuf, 4096)) == 0)
            sleep_msec(200);
        if (k < 0)
            E_FATAL("cont_ad_read failed\n");

        /*
         * Non-zero amount of data received; start recognition of new utterance.
         * NULL argument to uttproc_begin_utt => automatic generation of utterance-id.
         */
        if (uttproc_begin_utt (NULL) < 0)
            E_FATAL("uttproc_begin_utt() failed\n");
       uttproc_rawdata (adbuf, k, 0);
       printf ("Listening...\n"); fflush (stdout);
```

Anatomy of Live Decoder

```
/* Note timestamp for this first block of data */
ts = cont->read_ts;

/* Decode utterance until end (marked by a "long" silence, >1sec) */
for (;;) {
    /* Read non-silence audio data, if any, from continuous listening module */
    if ((k = cont_ad_read (cont, adbuf, 4096)) < 0)
        E_FATAL("cont_ad_read failed\n");
    if (k == 0) {
        /*
         * No speech data available; check current timestamp with most recent
         * speech to see if more than 1 sec elapsed.  If so, end of utterance.
         */
        if ((cont->read_ts - ts) > DEFAULT_SAMPLES_PER_SEC)
            break;
    } else {
        /* New speech data received; note current timestamp */
        ts = cont->read_ts;
    }
```

Anatomy of Live Decoder

```
/*
         * Decode whatever data was read above.  NOTE: Non-blocking mode!!
         * rem = #frames remaining to be decoded upon return from the function.
         */
        rem = uttproc_rawdata (adbuf, k, 0);

        /* If no work to be done, sleep a bit */
        if ((rem == 0) && (k == 0))
            sleep_msec (20);
    }

    /*
     * Utterance ended; flush any accumulated, unprocessed A/D data and stop
     * listening until current utterance completely decoded
     */
    ad_stop_rec (ad);
    while (ad_read (ad, adbuf, 4096) >= 0);
    cont_ad_reset (cont);

    printf ("Stopped listening, please wait...\n"); fflush (stdout);
```

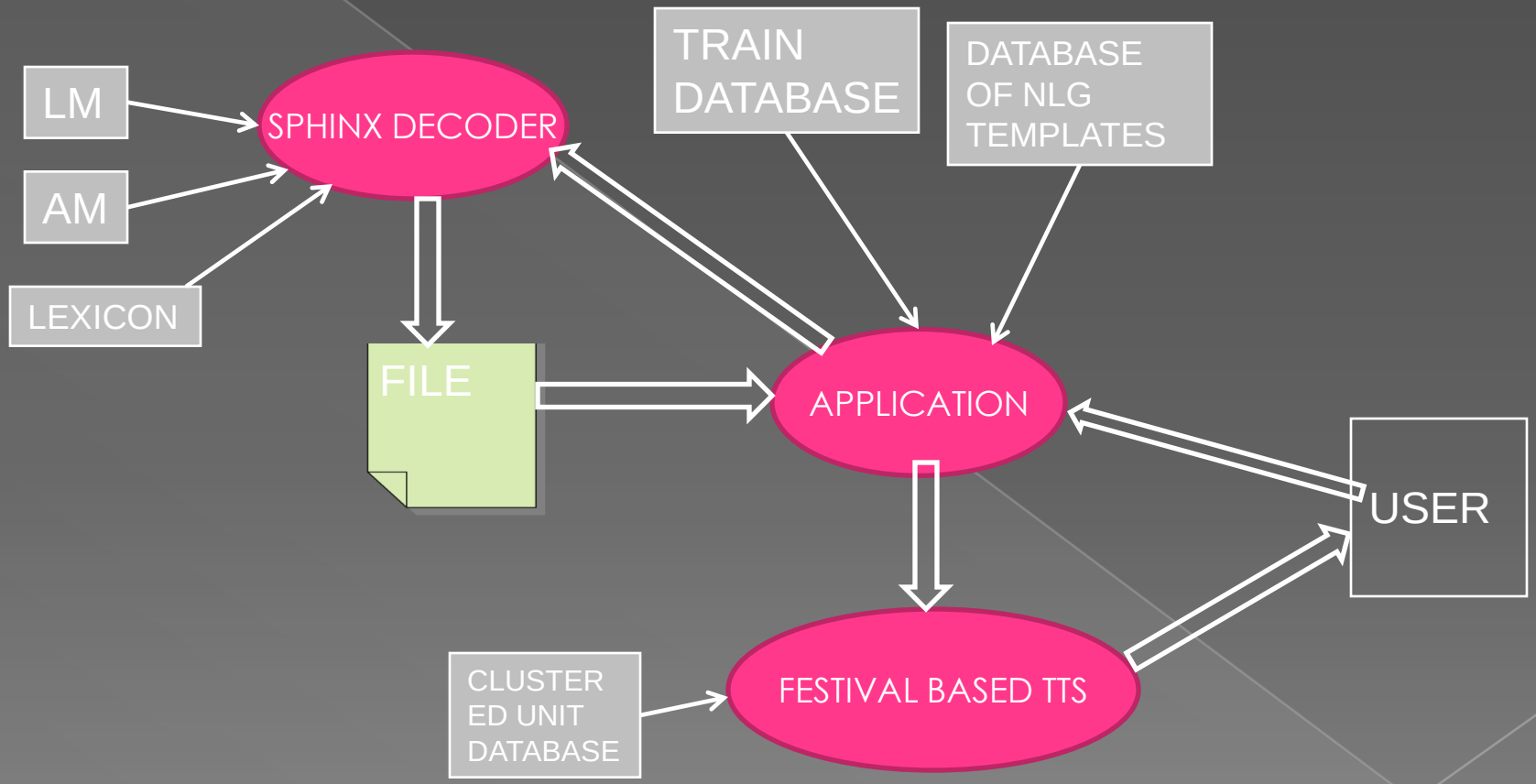
Anatomy of Live Decoder

```
#if 0
        /* Power histogram dump (FYI) */
        cont_ad_powhist_dump (stdout, cont);
#endif
        /* Finish decoding, obtain and print result */
        uttproc_end_utt ();
        if (uttproc_result (&fr, &hyp, 1) < 0)
            E_FATAL("uttproc_result failed\n");
        printf ("%d: %s\n", fr, hyp); fflush (stdout);

        /* Exit if the first word spoken was GOODBYE */
        sscanf (hyp, "%s", word);
        if (strcmp (word, "goodbye") == 0)
            break;

        /* Resume A/D recording for next utterance */
        if (ad_start_rec (ad) < 0)
            E_FATAL("ad_start_rec failed\n");
    }

    cont_ad_close (cont);
}
```

Railway Dialog System

- Ask train list from source to destination
  - मुंबई से बेंगलोर जानेवाली सारी ट्रेने बताओ
  - चेन्नई से दिल्ली जानेवाली ट्रेन्स् की लीस्ट दो
  - हैद्राबाद से मुंबई की ओर जाने के लिये शाम को कौनसी ट्रेन्स् अव्हेलेबल है
- Ask timings of a particular train
  - हुस्सैन सागर एक्सप्रेस का टाईम बताओ
- Built on Loosely Coupled Architecture
- Uses Blocking Mode

Architecture of train dialog system



ACKNOWLEDGEMENTS

- Venkatesh Keri
- Sriram Chaudhury
- Vinay Pamarthi

- Thanks to Sarai !