Developer Documentation

SoniTalk

Ultrasonic communication (UC) is increasingly used for data exchange between mobile phones and other devices, as well as for location-based services. UC is attractive because it is inaudible and very low-threshold in terms of the hardware required (only microphone and speaker required). Today, there exist several proprietary solutions for UC on the market, which are developed by companies in a closed source form, which raises questions regarding the protection of users' privacy, since it is not clear which data is actually sent, when and to whom.

SoniTalk is a novel open and transparent protocol for ultrasonic communication between devices such as smartphones, TVs, and IoT devices. Thereby SoniTalk gives the user full control over her privacy by a fine grained permission system.

The project was pursued by Matthias Zeppelzauer, Alexis Ringot and Florian Taurer at the Media Computing Group of the University of Applied Sciences in St. Pölten. In the future, SoniTalk could benefit various industries as well as end consumers with new communication possibilities.

What to find in this document

This documentation is aimed at developers willing to extend and/or adapt the SoniTalk SDK to their needs. You will find here additional information on the SDK architecture.

License

The SoniTalk SDK is licensed under LGPL. The SoniTalk Demo app is licensed under GPL.

Source Code

Available at https://github.com/fhstp/SoniTalk

Contributing

See https://github.com/fhstp/SoniTalk

Dependencies

You can see the libraries used and their licences at https://github.com/fhstp/SoniTalk

Credits/Acknowledgment

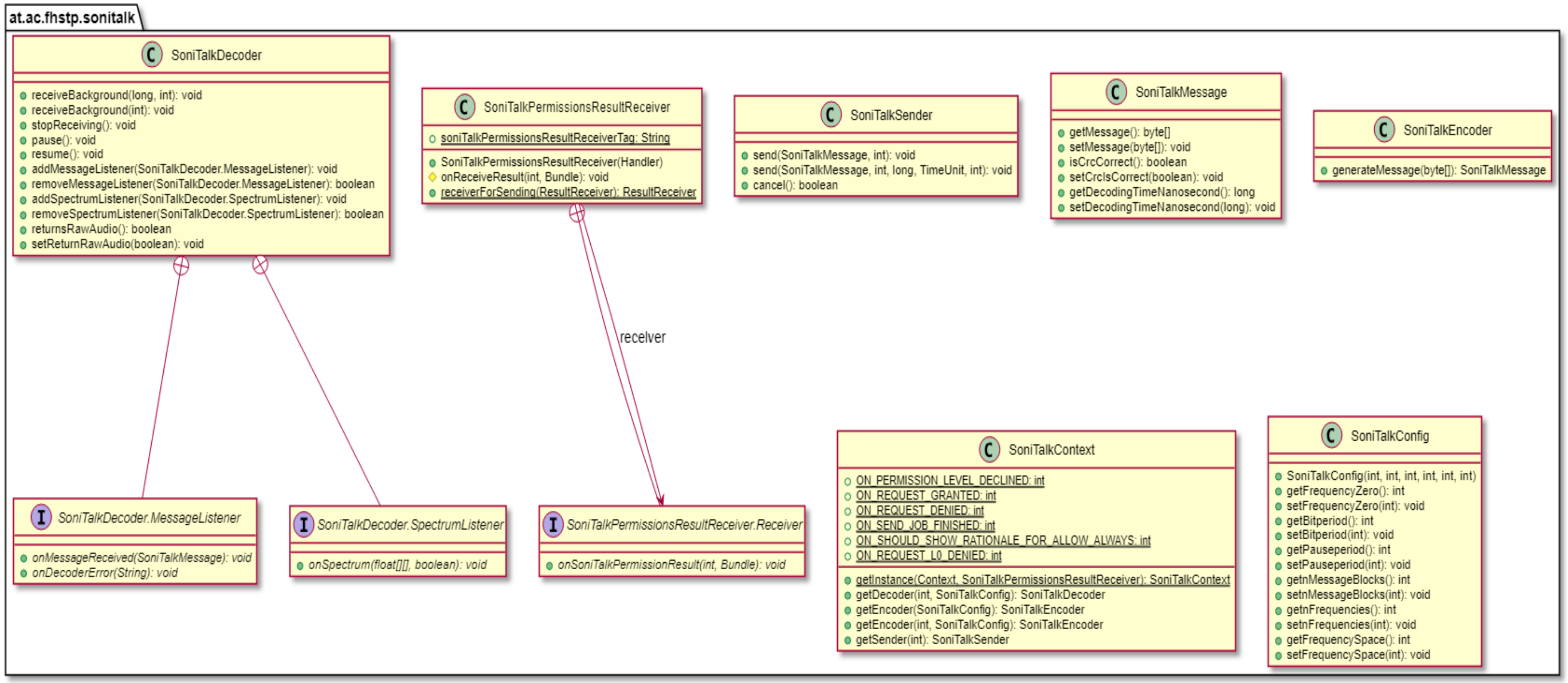
- Project funded by netidee
- SoniTalk is a project of the Institute for Creative\Media/Technologies (IC\M/T), developed at Sankt Pölten University of Applied Sciences (FHSTP)

SDK Setup

For information on prerequisites, how to include and how to use the SDK, please see: https://github.com/fhstp/SoniTalk#getting-started

Software Architecture and Implementation Details

Overview


at.ac.fhstp.sonitalk
SoniTalkDecoder
receiveBackground(long, int): void
receiveBackground(int): void
stopReceiving(): void
pause(): void
resume(): void
addMessageListener(SoniTalkDecoder.MessageListener): void
removeMessageListener(SoniTalkDecoder.MessageListener): boolean
addSpectrumListener(SoniTalkDecoder.SpectrumListener): void
removeSpectrumListener(SoniTalkDecoder.SpectrumListener): boolean
returnsRawAudio(): boolean
setReturnRawAudio(boolean): void
SoniTalkPermissionsResultReceiver
soniTalkPermissionsResultReceiverTag: String
SoniTalkPermissionsResultReceiver(Handler)
onReceiveResult(int, Bundle): void
receiverForSending(ResultReceiver): ResultReceiver
SoniTalkSender
send(SoniTalkMessage, int): void
send(SoniTalkMessage, int, long, TimeUnit, int): void
cancel(): boolean
SoniTalkMessage
getMessage(): byte[]
setMessage(byte[]): void
isCrcCorrect(): boolean
setCrcIsCorrect(boolean): void
getDecodingTimeNanosecond(): long
setDecodingTimeNanosecond(long): void
SoniTalkEncoder
generateMessage(byte[]): SoniTalkMessage
receiver
SoniTalkDecoder.MessageListener
onMessageReceived(SoniTalkMessage): void
onDecoderError(String): void
SoniTalkDecoder.SpectrumListener
onSpectrum(float[][], boolean): void
SoniTalkPermissionsResultReceiver.Receiver
onSoniTalkPermissionResult(int, Bundle): void
SoniTalkContext
ON_PERMISSION_LEVEL_DECLINED: int
ON_REQUEST_GRANTED: int
ON_REQUEST_DENIED: int
ON_SEND_JOB_FINISHED: int
ON_SHOULD_SHOW_RATIONALE_FOR_ALLOW_ALWAYS: int
ON_REQUEST_L0_DENIED: int
getInstance(Context, SoniTalkPermissionsResultReceiver): SoniTalkContext
getDecoder(int, SoniTalkConfig): SoniTalkDecoder
getEncoder(SoniTalkConfig): SoniTalkEncoder
getEncoder(int, SoniTalkConfig): SoniTalkEncoder
getSender(int): SoniTalkSender
SoniTalkConfig
SoniTalkConfig(int, int, int, int, int, int)
getFrequencyZero(): int
setFrequencyZero(int): void
getBitperiod(): int
setBitperiod(int): void
getPauseperiod(): int
setPauseperiod(int): void
getnMessageBlocks(): int
setnMessageBlocks(int): void
getnFrequencies(): int
setnFrequencies(int): void
getFrequencySpace(): int
setFrequencySpace(int): void

UMLDoclet 1.1.3, PlantUML 1.2018.12

The SoniTalk SDK core classes are SoniTalkContext, SoniTalkPermissionManager, SoniTalkConfig, SoniTalkMessage, SoniTalkEncoder, SoniTalkDecoder and SoniTalkSender.

Permission system

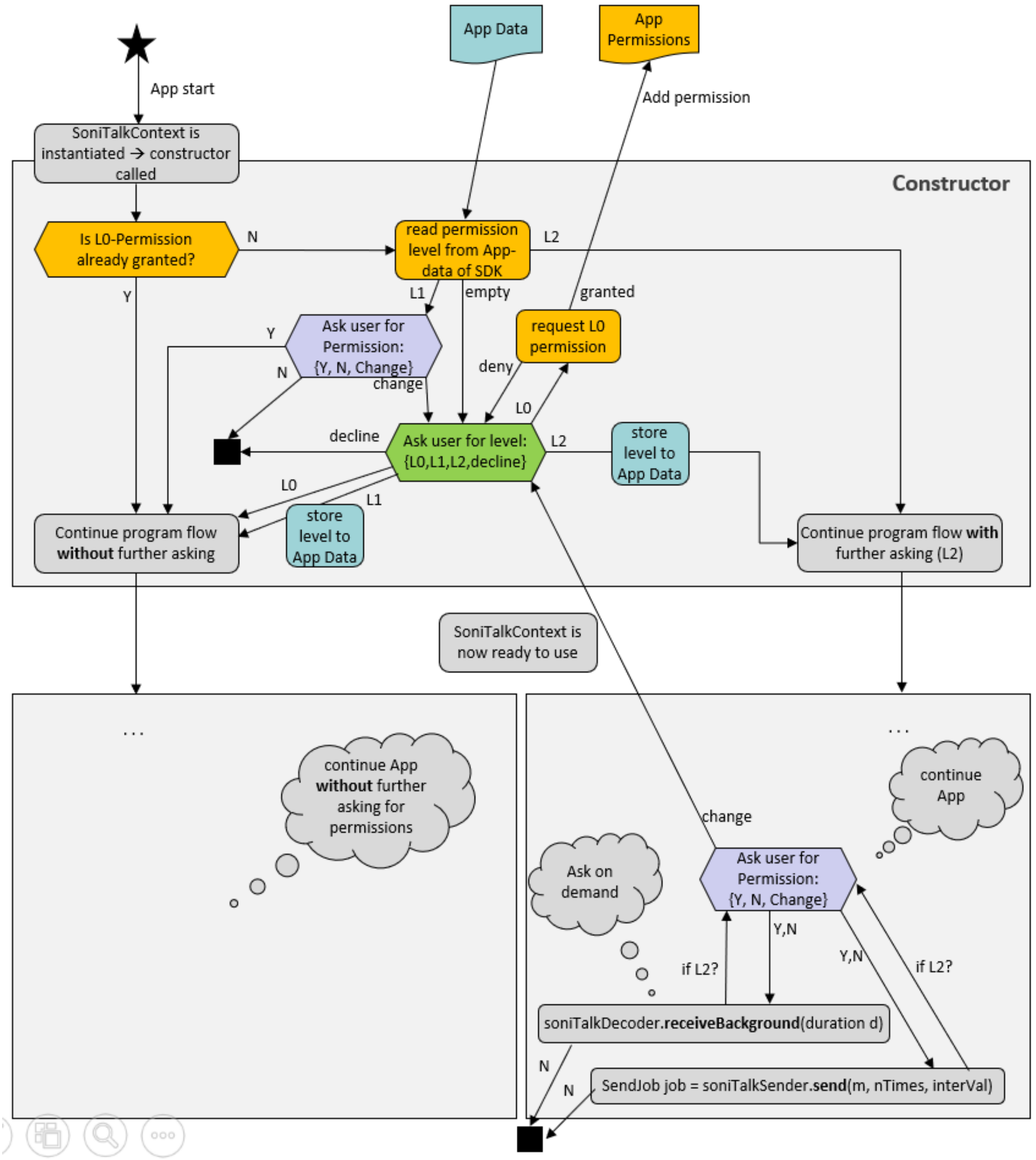
SoniTalk includes a built in permission and notification system that ensures the user always knows if an app is currently receiving/sending data via SoniTalk. Depending on the privacy level chosen by the user, permission will be granted for one communication at a time or for longer.

There are three privacy levels:

- **Low** - Allow always (called Level 0 or **L0**)
  → The user will be prompted with an Android permission dialog the first time the app tries to use SoniTalk.
- **Moderate** - Allow until next start of app (called Level 1 or **L1**)
  → The user will be prompted with a custom dialog after each app start when trying to use SoniTalk.
- **Strict** - Ask on each communication request (called Level 2 or **L2**)
  → The user will be prompted with a custom dialog for each communication request via SoniTalk.

Notes:

- Level 0 is implemented by a custom Android permission, otherwise it could not be revoked any more by the user.
- Levels 1 and 2 are handled internally, their dialog offers the option to change the privacy level.


App Data
App Permissions
App start
Add permission
SoniTalkContext is instantiated → constructor called
Constructor
Is L0-Permission already granted?
N
read permission level from App-data of SDK
L2
Y
L1
empty
granted
Ask user for Permission: {Y, N, Change}
Y
N
change
request L0 permission
deny
L0
decline
Ask user for level: {L0,L1,L2,decline}
L2
store level to App Data
L0
L1
store level to App Data
Continue program flow without further asking
Continue program flow with further asking (L2)
SoniTalkContext is now ready to use
...
continue App without further asking for permissions
...
continue App
change
Ask on demand
Ask user for Permission: {Y, N, Change}
Y,N
Y,N
if L2?
if L2?
soniTalkDecoder.receiveBackground(duration d)
N
N
SendJob job = soniTalkSender.send(m, nTimes, interVal)

For more information about the permission system, see SoniTalkPermissionManager

SoniTalkPermissionManager (at.ac.fhstp.sonitalk)

The SoniTalkPermissionManager ensures that SoniTalkDecoder and SoniTalkSender objects have the right permission before they start communicating.

Workflow

Before any action, SoniTalkDecoder and SoniTalkSender objects will start a new Thread and call `SoniTalkPermissionManager.checkSelfPermission(Context context, int` `requestCode)`. This call is synchronous and will return once the user answered the permission dialog(s). The first step is to check if the permission level was set and if it is not to ask for it synchronously in `PermissionLevelDialogActivity`.

```
if (checkPermissionLevelIsSet(context)) {
    return checkPermissionGranted(context);
}
else { // Permission level request was declined
    return false;
}
```

Once the permission level is set, we check if the permission to use data-over-sound is granted. If L0 was chosen, we check if the Android permission is granted (it was asked directly after the user chose the level 0).
If L1 or L2 was chosen, we ask for permission via the PermissionRequestDialogActivity.

Callbacks

`SoniTalkPermissionManager` implements two interfaces in order to get callbacks from the permission dialogs:

- PermissionLevelDialogActivity.PermissionLevelDialogListener
  ```
  {
          public void onL0Click(DialogFragment dialog);
          public void onL1Click(DialogFragment dialog);
          public void onL2Click(DialogFragment dialog);
          public void onDeclineClick(DialogFragment dialog);
   }
  ```
- PermissionRequestDialogActivity.PermissionRequestDialogListener)
  ```
  {
          public void onGrantClick(DialogFragment dialog);
          public void onDenyClick(DialogFragment dialog);
          public void onChangeSettingsClick(DialogFragment
  dialog);
  }
  ```

These callbacks persist some flags (e.g. if the request was answered/granted) and then unlock the execution of the functions that were locked after calling the dialog (e.g. checkPermissionGranted). The flags are then retrieved and execution continues. The corresponding callbacks are sent to the developer via a `ResultReceiver sdkListener`, that is provided by the developer's app. These callbacks are defined in the SoniTalkContext class. See the Javadoc or the SoniTalkContext chapter of this documentation for more information.

ResultReceiver was used as it is one of the built in Android way of passing a listener to a service via a Bundle. Similarly, SoniTalkPermissionManager was made Parcelable in order to be passable to the permission dialog activities.

It was decided to use transparent activities showing standard alert dialogs because we needed to keep control over the permission choices while not disturbing the user experience.

SoniTalkContext (at.ac.fhstp.sonitalk)

The SoniTalkContext allows to create objects of the Encoder, Decoder and Sender. It handles the creation, initialization, updated and cancelation of notifications. Beside that the SoniTalkContext contains the request codes for the permission levels, granting and denying permission requests and checking if a send job is finished.

For more details on the callbacks see the Javadoc

SoniTalkConfig (at.ac.fhstp.sonitalk)

The SoniTalkConfig class handles the configuration, or profile, used to transmit data. The emitter and receiver of a message must use the same configuration. Please see the SoniTalk protocol specification document for more details on the parameters.

A crucial use case will be transmitting with several profiles simultaneously. This will allow for faster communication within one app and simultaneous communication of several apps.

ConfigFactory (at.ac.fhstp.sonitalk.utils)

Utility class for building SoniTalkConfig objects from JSON files. Please see the Javadoc for more details.

SoniTalkMessage (at.ac.fhstp.sonitalk)

SoniTalkMessage contains the message itself (as a byte array), the raw audio data as a short array, and debugging variables (a boolean indicating if the CRC is correct and the time of decoding).

SoniTalkDecoder (at.ac.fhstp.sonitalk)

The SoniTalkDecoder handles the capture of audio, the detection of messages and their decoding.

When creating an audiorecorder with getAudioRecorder(), the buffersize has to be at least bigger than the minimum buffer size of Android internal AudioRecorder. We set the initial buffer size depending on the analysis window length (the latter is currently half of the bitperiod) and increase to the minimum buffer size it if is too small.

```
minBufferSize = AudioRecord.getMinBufferSize(Fs,
        AudioFormat.CHANNEL_IN_MONO,
AudioFormat.ENCODING_PCM_16BIT);

if (audioRecorderBufferSize < minBufferSize) {
   audioRecorderBufferSize = minBufferSize;
}
```

Audio is captured one “analysisWindowStep” at a time, this step can be e.g. 8 times smaller than the analysis window. This avoids missing a message start. The analysis window length is half the length of a bit period (one fourth of a message block).

The main part of the decoder is the analysis of the history buffer. To check if the audio data contains a message, the first and the last window of the buffer are evaluated.

```
float firstWindow[] = new float[analysisWinLen];
float lastWindow[] = new float[analysisWinLen];
System.arraycopy(analysisHistoryBuffer, 0, firstWindow, 0,
analysisWinLen);
System.arraycopy(analysisHistoryBuffer,
analysisHistoryBuffer.length - analysisWinLen, lastWindow, 0,
analysisWinLen);
```

Both windows are copied twice (to compare the lower band with the upper band) and a bandpass filter is applied on all four arrays.

```
Butterworth butterworthDown = new Butterworth();
butterworthDown.bandPass(bandPassFilterOrder,Fs,centerFrequenc
yBandPassDown,bandpassWidth);

Butterworth butterworthUp = new Butterworth();
butterworthUp.bandPass(bandPassFilterOrder,Fs,centerFrequencyB
andPassUp,bandpassWidth);

for(int i = 0; i<startResponseLower.length; i++) {
   startResponseUpperDouble[i] =
     butterworthUp.filter(startResponseUpper[i]);
   startResponseLowerDouble[i] =
     butterworthDown.filter(startResponseLower[i]);
}
```

A hilbert transformation is then applied and the total energy of the upper and lower half are compared. A start block is detected if the high frequencies are greater than the low frequencies with a factor of startFactor (usually 2). An end block is detected if the low frequencies are greater than the high frequencies with a factor of endFactor (usually 2).

```
if(sumAbsStartResponseUpper > startFactor *
sumAbsStartResponseLower)

if(sumAbsEndResponseLower > endFactor *
sumAbsEndResponseUpper)
```

If both a start block and an end block were detected in the buffer, we move on to decoding. The analyzing of the message part itself, includes appling of a hamming window, fft, normalization, logarithmization and cutting away unimportant frequencies. Then the centers of the message blocks and their frequency indices are calculated to decode the Manchester encoded bits. Basically, high followed by low (energy) is a 1, and low followed by high (energy) is a 0.

For more details on the encoding / decoding specification, please see our Specification document.

The receiveBackground functions execute in a worker Thread and need to be stopped when the application stops.
When receiving is done, the resources, like microphone, will be released in stopReceiving()

```
public void stopReceiving() {
   setLoopStopped(true);

   soniTalkContext.cancelNotificationReceiving();

   delayhandler.removeCallbacksAndMessages(null);

   ...
}
```

SoniTalkEncoder (at.ac.fhstp.sonitalk)

The SoniTalkEncoder encodes the forwarded byte array and uses a SignalGenerator to get the raw audio data as short array for the SoniTalkMessage. This audio data is then concatenated to have the right shape for creating an audio track and sending it.

The byte array gets encoded via the encoderUtils and delivers a String of bit. To have the correct size of the encoded data String, the number of message blocks is needed.

```
String bitOfText = encoderUtils.getStringOfEncodedBits(data,
config.getnMessageBlocks());
```

CreateStringArrayWithParityOfBitText transforms the String into a String array and calculates and attaches a parity sequence to the original array. Therefore a generator polynom is needed.

```
createStringArrayWithParityOfBitText(...)
```

Important is to create an inverted version of the String array. Both are then used to generate the empty containers for the signal and fill with audio data. Depending on the number of blocks two empty two-dimensional arrays are created and filled with the 0s and 1s of the message.

```
for (int i = 0; i < numberOfFrequencies; i++) {
   messageSplitted[j][i] =
bitStringArray[i+(numberOfFrequencies*j)];
   messageSplittedInverted[j][i] =
bitStringArrayInverted[i+(numberOfFrequencies*j)];
}
```

Corresponding to those two String arrays, two two-dimensional short arrays are created to be filled with the audio data. They consist of three different parts:
The startblock which is filled with 1s in the upper half,

```
String[] protoArrayStart = new String[numberOfFrequencies];
```

```
Arrays.fill(protoArrayStart, 0, (numberOfFrequencies/2)-1,
"0");
Arrays.fill(protoArrayStart, numberOfFrequencies/2,
numberOfFrequencies, "1");
short[] protoTrackStart =
signalGen.getSignalBlock(SignalType.PLAYCONFIG,
protoArrayStart);
```

the endblock which is filled with 1s in the lower half and

```
String[] protoArrayEnd = new String[numberOfFrequencies];
Arrays.fill(protoArrayEnd, 0, numberOfFrequencies/2, "1");
Arrays.fill(protoArrayEnd, numberOfFrequencies/2,
numberOfFrequencies, "0");
short[] protoTrackEnd =
signalGen.getSignalBlock(SignalType.PLAYCONFIG,
protoArrayEnd);
```

and the middle part which is representing the signal.

```
for(int k=0;k<mesLengthDividedNumFreq;k++) {
   frequencyZeroTrack[k] =
signalGen.getSignalBlock(SignalType.PLAYCONFIG,
messageSplitted[k]);
   frequencyZeroTrackInverted[k] =
signalGen.getSignalBlock(SignalType.PLAYCONFIG,
messageSplittedInverted[k]);
}
```

For the main part a for loop iterates through the String arrays to create an individual signal block for every message block. Additionally a pausetrack with only zeros is created. Everything is then concatenated in the method `concatenateSignalBlocks(...)`

SoniTalkSender (at.ac.fhstp.sonitalk)

See the Javadoc

DecoderUtils (at.ac.fhstp.sonitalk.utils)

The DecoderUtils includes the functions to remove filling characters and error checking bits, to cast the byte back to utf-8 characters and also to check for characters with more than one byte. Furthermore, methods for processing the received signal are provided.
For more information, see the Javadoc.

SignalGenerator (at.ac.fhstp.sonitalk.utils)

The SignalGenerator handles the creation of the audio message. It creates the blocks depending on the message forwarded. The generator includes indexing the frequencies, fft, normalization, fade-in/fade-out and casting to the right format.

The SignalGenerator creates a block of the audio message every time the method getSignalBlock is called.
Depending on the SignalType the correct window length will be selected. The bitperiod defines the length of message blocks and the start and end block, and the pauseperiod is for the breaks in between blocks.

```
if(signalType.equals(SignalType.PLAYCONFIG)) {
   winLen = config.getBitperiod();
}else if(signalType.equals(SignalType.PAUSECONFIG)) {
   winLen = config.getPauseperiod();
}
```

To be sure that the window length has the correct size for the later created audiotrack, we check that it is not under 30. The size of the samples calculated from the window length then has to be checked if it is odd or even, because the audiotrack of Android cannot use an odd samplesize, so one value will be added to make it even.

```
if (winLen==0){winLen= 30;}

winLenSamples = winLen*fs/1000;

if(winLenSamples%2 == 1){
   winLenSamples+=1;
}
```

To get all frequencies for one block the method useSignalConfig has to be called and will be saved in a two dimensional array with the name whiteNoiseBands and the signal type and the array with the bits as parameters.

```
whiteNoiseBands = useSignalConfig(signalType,
bitStringArray);
```

The method useSignalConfig has a switch case for the two signal types. Basically both check for zero and one in the forwarded String array. The playconfig takes the base frequency declared as frequencyZero and the space between every frequency as spaceBetweenfrequencies. Then a for loop with the length of the number of frequencies is started and iterates through the String array. Everytime the array equals zero it will be saved in an array, called mLine, as zero and if the array value equals one the frequency will be calculated. This happens as the current index of the for loop and the space between the frequencies will be multiplied and then added to the base frequency.

```
case PLAYCONFIG:
   int frequencyZero = config.getFrequencyZero();
   int spaceBetweenFrequencies = config.getFrequencySpace();

   for(int i = 0; i<numberOfFrequencies; i++) {
       if(bitStringArray[i]!=null) {
           if (bitStringArray[i].equals("0")) {
               mLine[freqCounter] = "0";
               freqCounter++;
           } else if (bitStringArray[i].equals("1")) {
```

```
                    mLine[freqCounter] =
String.valueOf((frequencyZero + (i *
spaceBetweenFrequencies)));
                    freqCounter++;
                }
            }
        }
        break;
```
In comparison to that the pauseconfig just fills the array with zeros.

As our SignalGenerator has bands with a higher and lower frequency border, we need to calculate those new frequencies with the help of the bandwidth. To achieve that a new two dimensional double array will be created. Then a for loop will iterate through the previously filled array mLine and check for zeros and ones again. Zeros will just be saved again as a zero for both borders. For ones, the bandwidth will be taken and divided by two to get the half. This half bandwidth will be then subtracted from the current frequency to get the lower border and to get the higher border, the half bandwidth will be added to the frequency. The index zero is always the lower border and the index one the higher one.

```
frequencyBands = new double[freqCounter][2];
for(int j = 0; j<freqCounter; j++) {
    if(mLine[j].equals("0")){
         frequencyBands[j][0] = 0;
         frequencyBands[j][1] = 0;
    }else {
         frequencyBands[j][0] = (Integer.parseInt(mLine[j]) -
(bandWidth / 2));
         frequencyBands[j][1] = (Integer.parseInt(mLine[j]) +
(bandWidth / 2));
    }
}
```

To get the right position within the signal samples, we have to get the indices of every band.
```
for(int i = 0; i<whiteNoiseBands.length; i++) {     double
freqDown = whiteNoiseBands[i][0];
    double freqUp = whiteNoiseBands[i][1];
    cutoffFreqDownIdx[i] = Math.round(freqDown / (fs / 2) *
(winLen * fs / 1000) + 1);
    cutoffFreqUpIdx[i] = Math.round(freqUp / (fs / 2) * (winLen
* fs / 1000) + 1);
}
```

Before we can use the fft we create a new double array with doubled the size of the samples. Which we then fill with the values of samples.

```
double[] complexSignal = new double[winLenSamples * 2];
```

```
System.arraycopy(inputSignal, 0, complexSignal, 0,
winLenSamples);
```

After calling the fft we take the lowest and the highest frequency and “cut” the signal by setting all values under and above those frequencies to zero. This happens twice as the complex signal is mirrored.
This sets everything under the lowest frequency to zero.

```
for (double j = 0; j < minFreq; j++) {
   complexSignal[(int)j] = 0.0f;
}
```

The same thing done for the mirrored part.

```
double helpWinLenSamples = winLenSamples * 2;
for (double j = (helpWinLenSamples - (minFreq-1)); j <
helpWinLenSamples; j++) {
   complexSignal[(int)j] = 0.0f;
}
```

This sets everything above the highest frequency to zero and also does it for the mirrored part.

```
double helpUpSamples = winLenSamples - (maxFreq+1);
for (double j = winLenSamples - helpUpSamples; j <
winLenSamples + helpUpSamples; j++) {
   complexSignal[(int)j] = 0.0f;
}
```

After the initial setup, a for loop with all bands is executed. Therefore, all the parts in between get also set to zero and again mirrored.

```
if(whiteNoiseBands.length>1) {
   for (int k = 0; k < whiteNoiseBands.length-1; k++) {
        for (double l = cutoffFreqUpIdx[k]+1; l <
cutoffFreqDownIdx[k+1]; l++) {
             complexSignal[(int)l] = 0.0f; //set all frequencies
between the higher frequency of one band to the lower
frequency of the next band to 0
        }
        int helpSamples = winLenSamples * 2;
        for (double l = helpSamples-cutoffFreqDownIdx[k+1]+1; l
< helpSamples-cutoffFreqUpIdx[k]; l++) {
             complexSignal[(int)l] = 0.0f; //set all frequencies
between the higher frequency of one band to the lower
frequency of the next band to 0 mirrored to the doubled
winLenSamples size
        }
   }

   for (int k = 0; k < whiteNoiseBands.length; k++) {
        for (double l = cutoffFreqDownIdx[k]; l <=
cutoffFreqUpIdx[k]; l++) {
```

```
                complexSignal[(int)l] = 1000;
            }
            int helpSamples = winLenSamples * 2;

            for (double l = helpSamples-cutoffFreqUpIdx[k]; l <=
    helpSamples-cutoffFreqDownIdx[k]; l++) {
                complexSignal[(int)l] = 1000;
            }
        }
    }
```

EncoderUtils (at.ac.fhstp.sonitalk.utils)

The main part of EncoderUtils is to change the message bytes to bit. Beside that it checks for characters with more bytes and offers a function to check if the number of bytes exceed a specific value

To get the correct bits out of the forwarded byte array, we have to check for characters with more than one byte. Therefore, we can check for the leading byte which has, depending on the amount of bytes, the same amount of ones in the first bits of the first byte.

```
    for(int i = 0; i <= bitOfText.length() - 8; i+=8)
    {
       if((i+32) <= bitOfText.length() && bitOfText.substring(i,
    i+3).equals("1111")) {
            int k = Integer.parseInt(bitOfText.substring(i, i +
    32), 2);
            outputByte[counter++] = (byte)k;
            i+=24;
       }else if((i+24) <= bitOfText.length() &&
    bitOfText.substring(i, i+2).equals("111")) {
            int k = Integer.parseInt(bitOfText.substring(i, i +
    24), 2);
            outputByte[counter++] = (byte)k;
            i+=16;
       }else if((i+16) <= bitOfText.length() &&
    bitOfText.substring(i, i+1).equals("11")) {
            int k = Integer.parseInt(bitOfText.substring(i, i +
    16), 2);
            outputByte[counter++] = (byte)k;
            i+=8;
       }else {
            int k = Integer.parseInt(bitOfText.substring(i, i + 8),
    2);
            outputByte[counter++] = (byte)k;
       }
    }
```

To always be sure that the number of bytes is not exceeding the provided maximum, we check with the method isAllowedByteArraySize.

```
public static boolean isAllowedByteArraySize(byte[]
textToSend, int nMessageblocks){
   int maxChars =  nMessageblocks*2-2;
   if(changeToBitString(textToSend).length()<=(maxChars*8)) {
       return true;
   }else{
       return false;
   }
}
```

CRC (at.ac.fhstp.sonitalk.utils)

The CRC class adds and also checks the parity bits. The main part of both functions is a xor_logic_gate. Beside that it has a helper function for counting occurrences. The message will be checked in a recursion with a generator polynomial until only zeros are left or the first element of the checked message is not a one anymore. The generator serves as a window which is slided through the message.

```
int j = 0;
int m = checkZeroMessage(byteMessage,j);
if(byteMessage.length-m<generatorPolynom.length){
   int p = 0;
   for(int o = byteMessage.length-(generatorPolynom.length-1);
o<byteMessage.length;o++){
        helpBitArray[p] = byteMessage[o];
        p++;
   }
}else{
   for(j = 0; j<generatorPolynom.length; j++){
        int xor = byteMessage[m] ^ generatorPolynom[j];
        byteMessage[m] = (byte)(0xff & xor);
        m++;
   }
   xorArray(byteMessage, helpBitArray);
}
```

For the sending part, the elements of the checked message, with a length of the generator polynomial minus one, will be added to the message as parity bits. The receiver part only checks the occurrence of ones in the checked message. If there are zero occurrences, the message is correct.