Developer Documentation

SoniControl is a novel technology for the recognition and masking of acoustic tracking information. The technology helps end-users to protect their privacy. Technologies like Google Nearby and Silverpush build upon ultrasonic sounds to exchange information. More and more of our devices communicate via this inaudible communication channel. Every device with a microphone and a speaker is able to send and receive ultrasonic information. The user is usually not aware of this inaudible and hidden data transfer. To overcome this gap SoniControl detects ultrasonic activity, notifies the user and blocks the information on demand. Thereby, we want to raise the awareness for this novel technology.

The project SoniControl is funded by Netidee (www.netidee.at) and is a project at the Media Computing Group at the Institute for Creative\Media/Technologies at Sankt Pölten University of Applied Sciences (mc.fhstp.ac.at).
The project website of the SoniControl project with all published results and resources can be found here: sonicontrol.fhstp.ac.at. The SoniControl App can be downloaded on Google Play Store.

License

Source Code

The entire source code can be found on: git.nwt.fhstp.ac.at/m.zeppelzauer/SoniControl

Contributing

Please feel free to open issues, submit pull requests, or just send us feedback at sonicontrol@fhstp.ac.at

Open topics / Features to add

- Recognition of the detected signal
- Record the signal and play it back in the audible range

Credits

- Audio by Superpowered (http://www.superpowered.com/)
- Material Icons, which are under Apache License Version 2.0 (www.apache.org/licenses/LICENSE-2.0.txt)
- The project SoniControl is funded by Netidee (www.netidee.at)

Installation & Setup

Sonicontrol is an Android application, developed in Java and C++ using Android Studio 3. We used the library SuperpoweredSDK (http://superpowered.com/) for the soundprocessing part.

To compile and run the project you need to:

- download the source code (git.nwt.fhstp.ac.at/m.zeppelzauer/SoniControl),
- import it in Android Studio version 3 or above,
- download the corresponding Android SDK and NDK,
- download the Superpowered SDK (http://www.superpowered.com/),
- link to Superpowered SDK in the local.properties file (at the root of the Android Studio project)
  e.g. : superpowered.dir=[some_path]/SuperpoweredSDK/Superpowered

The first version is usable on devices running Android 4.1 and above.

Software Architecture and Implementation Details

User Interface overview

Our application consists of three main screens/activities (main, settings and stored locations activities). The main activity has five buttons to: start / stop scanning, open the detected locations, open the settings, and the last one with the “x” icon to exit the app and release all resources.

Start of the app

When tapping on the start-Button, a service and a threadpool are created. We start our scan process in one thread. We also request location updates in order to have a precise location when the user detects a signal. This allows the app to remember where the user wants to block/ignore ultrasonic signals.

SoniControl Detector

The SoniControl Detector is implemented in C++ in “FrequencyDomain.cpp”, and called from the Java “Scan” class. The underlying concept is that we create a background model of the surrounding ultrasonic noise and then detect strong changes (signals).

Please have a look at the flowchart “SoniControl Detector” at the end of this document.

We use Superpowered to get the audio input with low latency and compute the fast Fourier transform (FFT). This transforms the signal from the time domain to the frequency domain (namely to a spectrogram), making it possible to evaluate the amplitude of the signal for each frequency.

The main steps of our processing are for each sample (every ~46ms):

- Filter the audible frequencies. We apply a highpass filter at about 17kHz in order to analyze only the ultrasounds (some technologies use frequencies at the edge of the hearable range, which is the reason for this rather low value),
- Normalize the spectrogram,
- Add this normalized spectrogram to the background buffer (which is a list of spectrograms),
- Check if the background buffer is full (after about 10s), if it is, we can start analyzing it as follow:
    - We compute the “current background model”, which contains for each frequency, the median amplitude value over the last 10s,
    - We compare this current background model to the current normalized spectrogram (using the Kullback Leibler Divergence as distance metric),
    - If the difference is high, we consider it as a “detection”, or rather a sub-detection as it is calculated on a rather short time (about 46ms),
    - We put this “detection” result (0 or 1) in a “median buffer”,
    - If this median buffer is full (after 1,5s), we compute its median, if it is 1 we consider that we detected an ultrasonic communication.
      (meaning that if over the last 1,5s there was more “detections” than “non-detections”, we consider there really was an ultrasonic communication)
        - If we detected something, we delete the last entries in the background model to avoid learning the detected signal as being normal.

Recognition of the signal

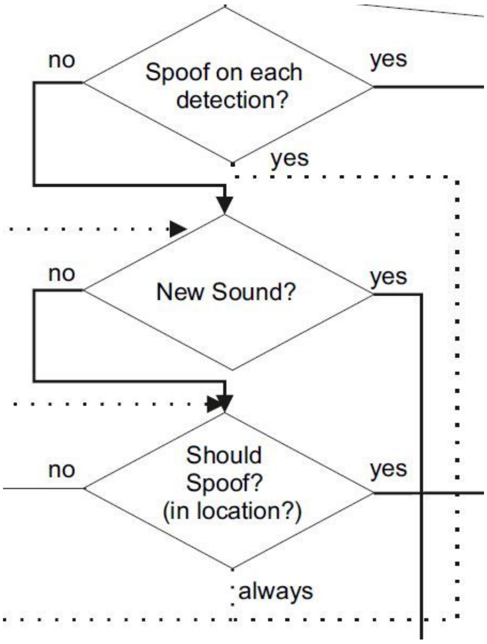
The recognition of different ultrasound transmission technologies and protocols will be subject of further development and is planned as a next step.

On signal detection

When a signal is detected, the current location is determined by a java class called “GPSTracker”, which handles the location methods like “getLongitude” and “getLatitude”, and cached until user choice.

Detections are then handled as follow:

- Spoof on each detection



After a Signal is found, we check the setting “Block on each location”. When this is checked, the user always wants to block detections, which will lead him to the blocking part (described in “Blocking routine”).

- New sound?

When “Block on each location” is unchecked, we will loop through the entries in the JSON-file, where all detections are saved, to check if the location of the current detection match a previous detection. The distance between the detection location and the one from the JSON entry is calculated to check if it is within the radius. The radius is a separate entry in the settings activity called “Location radius (x metres)”.

If there is a match, the process will lead to the question “Should spoof?”, which is described in the next paragraph.
If no entry matches the detected location, it is a new signal and will open the alert. In this case, there is another setting called “Preventive blocking”, which is described below.

- Should spoof ? (in location?)

If there was a match with the JSON-file (we already had the same kind of signal here), we check the spoofing-status attribute of the matching JSON-entry. If this parameter is true, the process will go to the blocking part. Otherwise it will start scanning again, as the signal should not be blocked. In both situations the JSON-entry will get updated.

- Preventive blocking

Preventive blocking is another setting in the app, which is triggered when the alert opens. When the setting is checked, the detected signal will be blocked (leads to the “Blocking routine” described later), even before a user decision in the alert, for as long as it takes the user to make a choice in the alert.

Alert

When the alert is opened, the user gets four options. The “Block always here” and “Block this time” buttons lead to the blocking part. The other two buttons are “Dismiss this time” and “Dismiss always here”, they make it possible for the user to utilize the ultrasonic communication technology. One is only for one time and saved in a separate JSON-Object, while the other option is for permanently ignoring this signal at this location.

Blocking routine

Mic / Jam

There are two options for blocking. One is the active part, which will send out a broadband noise in the ultrasonic frequency area, and the other option is to block the microphone, so that no other app can use it. The Android OS only allows one app at the same time to use the microphone.
The routine starts with a check of the microphone access. If “Use the microphone for blocking” setting is checked and if the microphone is available, we start a new Audio Recorder to block the microphone. There will be nothing saved during this blocking part. When we do not have access to the microphone, we start jamming and sending out the noise. Both blocking actions update the notification.

Looping system

These two blocking methods run as long as specified in the setting “Blocking duration”. Then there will be a check for the location again, to see if we are still in the area of the detected signal. If we are, we start the routine again with checking for microphone access. If not, we start the scan and detection process again.

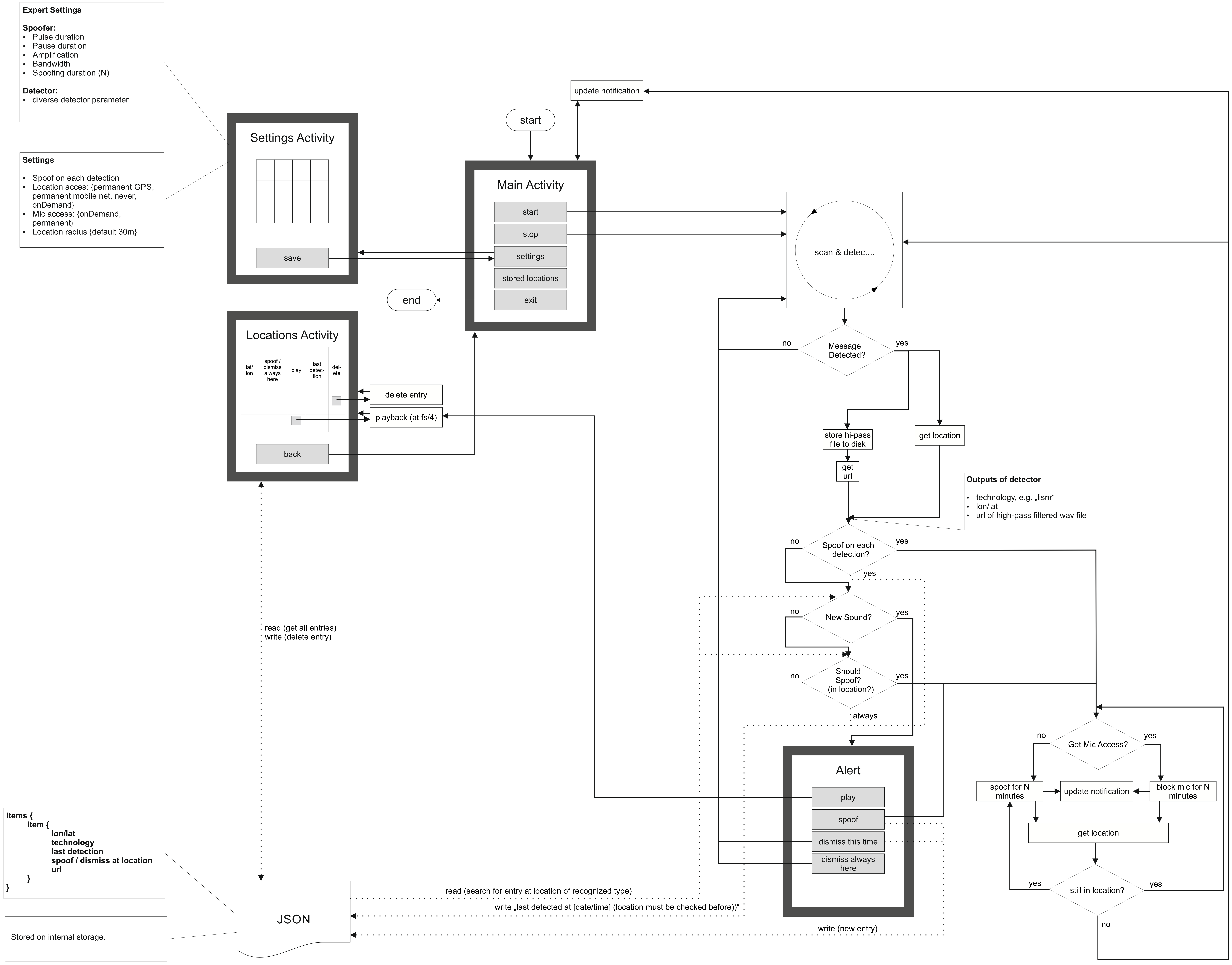
Stored Detections activity and JSON entries

Every detection will be saved into a JSON-file. In the “Stored Detections” activity we display all the detections where a location was found and the user chose an “always”-option.
The list is a listview item with a custom adapter and a custom row item, where all the information is filled into textviews and can be changed.

The JSON-file itself consists of three different JSON-Arrays which are:

- One for all found signals with correct location, which should be always blocked or always ignored,
- The second one is for all signal with correct location, but where the user decided one of the two “this time”-options at the alert,
- The last one is for all detected signals with no location data.

SoniControl System Architecture



Expert Settings

Spoofer:
- Pulse duration
- Pause duration
- Amplification
- Bandwidth
- Spoofing duration (N)

Detector:
- diverse detector parameter

Settings

- Spoof on each detection
- Location acces: {permanent GPS, permanent mobile net, never, onDemand}
- Mic access: {onDemand, permanent}
- Location radius {default 30m}

Settings Activity

save

start

update notification

Main Activity

start

stop

settings

stored locations

exit

end

Locations Activity

lat/ lon	spoof / dismiss always here	play	last detec- tion	del- ete

back

delete entry

playback (at fs/4)

scan & detect...

no

Message Detected?

yes

store hi-pass file to disk

get url

get location

Outputs of detector

- technology, e.g. „lisnr"
- lon/lat
- url of high-pass filtered wav file

no

Spoof on each detection?

yes

yes

no

New Sound?

yes

no

Should Spoof? (in location?)

yes

always

read (get all entries)
write (delete entry)

Alert

play

spoof

dismiss this time

dismiss always here

no

Get Mic Access?

yes

spoof for N minutes

update notification

block mic for N minutes

get location

yes

still in location?

yes

no

Items {
item {
lon/lat
technology
last detection
spoof / dismiss at location
url
}
}

JSON

Stored on internal storage.

read (search for entry at location of recognized type)

write „last detected at [date/time] (location must be checked before))"

write (new entry)

SoniControl Detector

Detection Parameters

Buffer sizes:
- bufferSize=50; %ms
- backgroundBufferSize = 10; %sec
- medianBufferSize = 1; %sec

Detector parameters
- cutoffFrequency = 16800; %lower limit for prontoly!
- decisionThreshold = 0.5; %this is for Kullback Leibler Divergence
- decisionThresholdNearby = 3.5; %the decision threshold for nearby signals. If the RMS Energy in the Nearby band is N-times higher than in the neighboring bands above and beneath the Nearby band, then declare a nearby detection.
- decisionThresholdNearbyAC = 0.05; %the recognition threshold for nearby based on autocorrelation (this is not used for detection, only later for recognition to differentiate nearby from other technologies like Lisner / Protntoly etc.

Recognition parameters
- specs.nearby.nBands=64;
- specs.nearby.bw=(20000-18500)./ specs.nearby.nBands/2; %unit Hz
- specs.nearby.centerFreq = 18496:23.6:20000;
- specs.lisnr.centerFreqs = [18750,18895,19051,19196,19500];
- specs.lisnr.bw = 40; %unit Hz
- specs.prontoly.centerFreqs=[16968,17054,17140,17226,17312,17398,17486,17571,17918,18430, 18516 ,18692,18778,18949,19035, 19379,19466,19724];
- specs.prontoly.bw = 10; %unit Hz, this is the minimum for prontoly
- specs.shopkick.centerFreqs=[19960,20040,20120,20200,20280,20360,20440,20520,20600,20680,20760,20840,20920,21000,21080,21160,21240,21320,21400,21480,21560,21640];
- specs.shopkick.bw = 4; %unit Hz
- specs.silverpush.centerFreqs=[18000,18075,18150,18225,18300,18375,18450,18525,18600,18675,18750,18825,18900,18975,19050,19125,19200,19275,19350,19425,19500,19575,19650,19725,19800,19875,19950];
- specs.silverpush.bw = 4; %unit Hz

start

Main Activity

start

stop

settings

stored locations

exit

Compute cutoffFrequencyIndex from cutoffFrequency (see function freq2idx.m)

Compute nFFT = number of expected FFT coefficients that remain after cutting away all frequencies below cutoffFrequencyIndex

Initialize all buffers (with zeros)

buffer: size = bufferSize

backgroundModelBuffer: size = [nFFT x backgroundBufferSize]

medianBuffer: size = medianBufferSize

bufferHistory: size = bufferSize * medianBufferSize]

this buffer stores the incoming audio signal for the total duration of the medianBuffer (e.g. 1.5s)

We use this longer signal for the detection of the type of message detected

This buffer is 1D

Initialize decision variables:
detection=0; %becomes one, if the current frame fulfulls detection condition
detectionAfterMedian = 0; %becomes one only if more than the half of the frames in the medianBuffer fulfill detection condition

Loop (until we have a detection)

buffer ← get next buffer

buffer ← convert buffer to mono (average over both channels)

bufferFFT ← abs(fft(buffer))

buffer_FFT_HiPass_Norm ← remove all coefficients below cutoffFrequencyIndex

buffer_FFT_HiPass_Norm ← buffer_FFT_HiPass_Norm/ sum(buffer_FFT_HiPass_Norm)

Is backgroundModelBuffer full?

no

yes

save current buffer in bufferHistory

append samples of current buffer to bufferHistory

compute median of backgroundModelBuffer

get median

backgroundDist ← get Kullback Leibler Divergence between median of backgroundModelBuffer and bufferFFT_HiPass_Norm

get

Nearby RMS detection (not implemented yet):

get bufferFFT

normalize: bufferFFTNorm ← bufferFFT /(sum(bufferFFT)+eps

centerFreqIdx ← get nearby center frequency indices

rmsIn ← get RMS between lowest and highest nearby frequency

rmsOut ← get RMS outside nearby frequency band

scoreNearby ← (rmsIn/rmsOut)^2

if backgroundDist > decisionThreshold OR scoreNearby > decisionThresholdNearby

no

yes

detection ← 0

detection ← 1

update medianBuffer: medianBuffer ← detection

push

Is backgroundModel Buffer and medianBuffer full?

no

yes

indivudial short detections (outliers) may become part of the background model

get median from medianBuffer

median is 1?

no

yes

to remove potential foreground sound in the background model)

update backgroundBuffer with bufferFFT_HiPass_Norm

clean backgroundBuffer: replace all items captured during the last medianBufferSize seconds by the values captured before

update

update

detected technology

get location