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: https://github.com/fhstp/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

- Support for other platforms. See iOS Concept

Credits

- Audio by Superpowered (https://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)
- Spectrogram visualization inspired from https://bitbucket.org/galmiza/spectrogram-android

Installation & Setup

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

To compile and run the project you need to:

- download the source code from https://github.com/fhstp/SoniControl,
- import it in Android Studio version 3 or above,
- download the corresponding Android SDK and NDK,
- download the Superpowered SDK (https://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
- click run and build the application for testing

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

Software Architecture and Implementation Details

User Interface overview

Our application consists of three activities (main activity, settings activity and firewall rules/detection history activity). The main activity has four buttons to: “start/pause” scanning, “stop” all processes/release resources, open the “Rules&Detections” activity, open the “settings” activity.

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). For an overview, 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 FFT 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 follows:
    - compute the “current background model”, which contains for each frequency, the median amplitude value over the last 10s,
    - compare this current background model to the current normalized spectrogram (using the Kullback Leibler Divergence as distance metric),
    - If the difference is high, 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 2,5s), we compute its median, if it is 1 we consider that we detected an ultrasonic communication
      (meaning that if over the last 2,5s there was more “detections” than “non-detections”, we consider there really was an ultrasonic communication)
        - If the “extended diagnostics” option is checked, a detection is only triggered once the signal is over (if the last 10% of the median buffer have 75% of non-detection). The idea behind it is to capture the whole signal for analysis and diagnostics purposes.
        - If we detected something, we delete the last entries in the background model to avoid learning the detected signal as being normal.

On signal detection

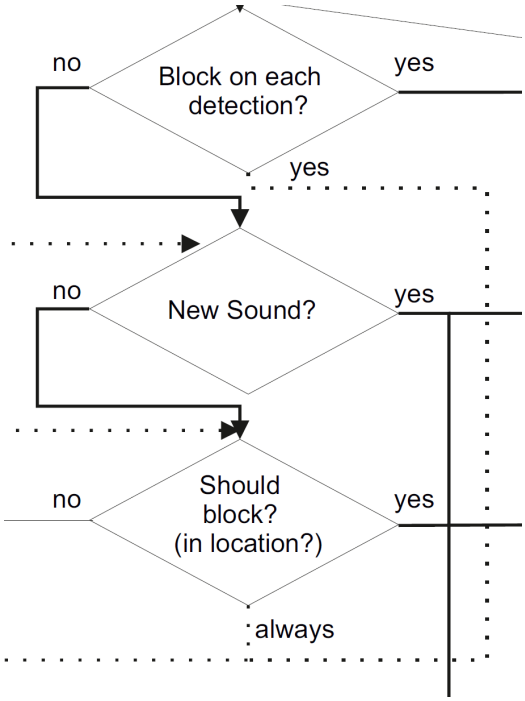
As soon as a signal is detected, the Preventive blocking option is triggered. If the setting is checked, the detected signal will be blocked (see “Blocking routine”) even before asking the user for a decision, and this blocking will be maintained until the user makes a choice in the alert dialog.

When a signal is detected, the second thing to happen is the preprocessing of the audio buffer, namely converting to mono and applying a high pass filter to remove all audible content. We then store this buffer for later use by the diagnostics module and the recognition process, described further in this document. The current location is then retrieved by a java class called "*GPSTracker*", which handles the location methods like "*getLongitude*" and "*getLatitude*", and caches this data until the user decides what to do with the signal detected.

Detections are then handled following this activity diagram:

Block on each detection?

The first step is to check the setting "Block on each location": if it is checked, all signals must be blocked which leads to the blocking process, 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 block?", which is described in the next paragraph. If no entry matches the detected location, it is a new signal and will open the alert asking the user to decide what to do with the signal.

Should block? (in location?)

If there was a match within the JSON-file (so if we already had the same kind of signal here), we check the blocking-status attribute of the matching JSON-entry. If the status is "Blocked", the process will go to the blocking part. If it is "Ask again", the user will be asked to decide what to do with the signal. If it is "Allowed", the firewall will start scanning again.

In all three situations the JSON-entry will get updated with the number of detections.

Detection AlertDialog

When the detection alert dialog opens, an instance of *DetectionDialogFragment* is created and a *DetectionAsyncTask* is started to adapt the UI (hide spectrogram placeholder and replay button, show a loading symbol) while processing the signal in the background. Once the

spectrogram is computed and the ultrasonic wave file created, the spectrogram and replay button are shown to the user, and the loading symbol is hidden. The user can then use Sonification to listen to a hearable pitch shifted version of the ultrasonic signal, as described further in this document.

The dialog offers the user with two main options to deal with the detected signal:

- **Block** (see Blocking routine further in this document), or
- **Allow** (making it possible for the user to utilize ultrasound to communicate if they want to).

Additionally, two checkboxes give “long-term” options regarding the signal for the user to :

- “**Make it available to the community**” (see Sharing functionality further in this document), and/or to
- “**Save it as a firewall rule (remember)**” which stores the detection in the corresponding JSON array in order to later be able to automatically block or allow the signal when detecting it at this place again (this decision can then be changed in the “Rules & Detections” activity).

Blocking routine

Block Microphone or Actively Jam

There are two options for blocking. One is the active part, which will send out white noise in the ultrasonic frequency area, and the other is to block the microphone, so that no other app can use it. Indeed, the Android OS only allows one app at the same time to use the microphone. If “Use the microphone for blocking” setting is checked, the routine checks the microphone access: if the microphone is available, we start a new Audio Recorder to block the microphone. No audio will be saved nor processed during this blocking part. If we do not have access to the microphone, we start actively jamming by sending out white noise. If the detected signal’s technology could be recognized, we only block its specific frequencies, otherwise the whole near-ultrasound range is jammed. Both blocking actions update the notification and status text to inform the user that blocking is ongoing.

Looping system

These two blocking methods would run as long as specified in the setting “Blocking duration”. The location will then be checked again to verify if we are still in the area of the detected signal. If we are, we start the routine again. If not, we start the scan and detection process again.

Diagnostics module

Recognition of the signal

The recognition of different ultrasonic communication technologies is mostly based on their characteristic frequencies. We studied signals from all technologies we knew about and analysed their spectral characteristics. We stored their frequency signatures in text files and implemented a recognition algorithm based on these. For each technology, we compute and

compare the on-band energy sum and the off-band energy sum. On-band means that this frequency band is “on”, i.e. it is used for transmission by this technology. Off-band means that this frequency band is “off”, i.e. it is not used for transmission by this technology. Here is the algorithm to compute these on and off band energy sums:

- Compute a Fast Fourier Transform (FFT) over the entire signal to get a spectrum (energy level at each frequency).
- Read the file containing the center frequencies into an array. This also gives you the number of on-bands and the number of off-bands (*nOnBands* + 1, as the first off-band starts from the cutoff frequency: 17kHz, and the last one ends at the maximum frequency for a standard microphone: 22.5kHz).
- For each band, loop through the spectrum at the corresponding frequencies and store the maximum energy[1].
- An off-band score is computed by averaging the stored maximum values: *offBandScore* = *sumOffBandEnergy* / *nOffBands*
- Similarly, an on-band score is computed, but it excludes the 25% lowest values as all frequencies/bands are not necessarily used for every signal.
- We return a score for this technology: the ratio *inBandScore*/*offBandScore*
- If the highest score is bigger than 1, the corresponding technology is shown to the user as “estimated type” detected, otherwise the uncertainty is too high and we declare the signal’s technology “unknown”.

Visualization

The spectrogram visualization is inspired from the class FrequencyView written by Guillaume Adam (see https://bitbucket.org/galmiza/spectrogram-android). Our *SpectrogramView* class displays a full spectrogram from a two dimensional float array spectrum. It is possible either to show all frequencies or a range between a lower and upper cutoff frequency.

The spectrogram computation starts in the *DetectionAsyncTask* class when the detection alert dialog creation is requested. During the creation of the *DetectionDialogFragment*, we configure the *spectrogramView* object with sampling rate, FFT resolution, (lower) cutoff frequency, upper cutoff frequency and the color theme to be used. If the computation is over already, we display the spectrogram, otherwise we show a loading symbol that will be hidden as soon as the *AsyncTask* background processing is done and the spectrogram shown.

Sonification

We use Pitch Shifting in order to make the detected signals hearable. The *PitchShiftPlayer* (SuperpoweredAdvancedAudioPlayer.h) class offers a simple pitch shifting audio player based on Superpowered library. A *PitchShiftPlayer* object is created when the button “Make it audible” is clicked in the detection alert dialog. The *MainActivity* implements the *PitchShiftPlayerListener* interface *onPlayCompleted* method which resets the button text once the pitch-shifted replay is over.

[1] Using the maximum value strongly decrease the score of other technologies, who get peaks in their off-frequencies. Taking the average or median would hide this pattern.

Rules & Detections activity and JSON entries

Every detection will be saved into a JSON-file. The JSON handling is capsuled in the *JSONManager* class. Within the JSON-file there are five JSON-arrays. These five are structured into one for all saved firewall rules, one for all one-time actions (block or allow) and one for all detections with an unknown location. The remaining two are for all imported firewall rules and for the history of all detections, where the first three JSON-arrays are combined into one array, independently of the user decision or location availability. Each JSON-entry consists of:

- Longitude
- Latitude
- Estimated type/technology of detection
- Spoofing status
- Address
- URL
- ID of estimated type/technology of detection
- Detection counter
- Amplitude

Besides getter, setter and deletion functionality for detections, update functions for the detection counter, the latest detection date and for the spoofing status are part of the class. Further, the *JSONManager* includes a sorting function for retrieving a chronological *ArrayList*.

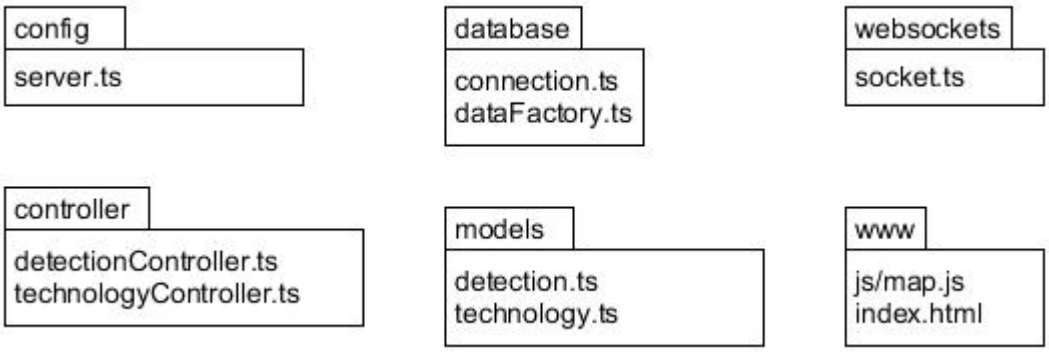
The activity for displaying those detections consist of four fragments: History, My Rules, Imported Rules and Map. Whereas the first three fragments show the corresponding JSON-arrays, the Map fragment displays an Open Street Map with marker for all saved rules, whether imported or personally saved ones.

Sharing functionality

For sharing detections with the community, the library Retrofit was used to communicate via REST. A *RESTController* class was implemented to connect to the server written in the *BASE_URL* variable, which is located in the local.properties-file. Further, the interface class *SoniControlAPI* was created with the REST calls for the endpoints on the server and a Detection class for the upload/download of detections. The upload functions for the detection data and the audio data are located in the *MainActivity*. The detection data is uploaded first and if this succeeds, the audio data is then also sent to the server.

Server

Now the server can be started! The server is based on Node.js with the extension Express.js for REST implementation and uses a MongoDB for data storage. It consists of six packages depicted in the figure below and further described in the listing below.



Packages:

- Starting with the *server.ts* of the **config package**, the basic objects get initialized and a connection to the database will be established. Further, the REST endpoints are defined here:
    - root (for map)
    - share
    - audioshare
    - getNumberOfImportDetections
    - importDetections
    - technologies
- The **controller package** includes a controller for detection handling and one for the technologies. The *detectionController* has the retrieving of detections implemented as well as filtering and grouping functionality, whereas the *technologyController* currently returns all technologies.
- The *connection.ts* of the **database package** includes the connection routine. Besides that, a *dataFactory* for the base data like technologies is implemented.
- Two **models** were created, one for detections and one for technologies.
- The **WebSocket** connections are located in the *socket.ts* and work as the interface for the map visualization.
- The **map** itself consists then of a html-file, a javascript-file and assets for showing the shared and uploaded detections on an Open Street Map.

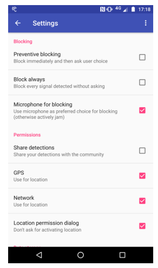
To start the server, either a self-hosted solution is needed, or a free hosting platform like Heroku. For hosting it on Heroku, a free account has to be created. There the Node.js application can be deployed via their Heroku CLI. Further, a MongoDB needs to be hosted somewhere. An example would be the free hosting plan at MongoDB. Next, on the Heroku dashboard of the created and deployed application, several config vars need to be entered. Those are also in the repository as dotenv-file. An example is given in the repository. The path

to the webfiles needs to be entered, as well as the port, rootpath and the database URL from MongoDB. All those entries are also needed for letting it run on a local machine. Therefore, the dotenv file needs to be copied, filled out and saved as '.env'. Last, the fileupload needs a kind of webspace or space on a server. It is done via sftp and the hostname, password and the user have to be written in the .env-file/Heroku config vars.

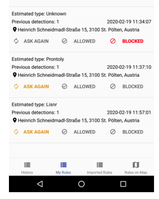
SoniControl System Architecture

see user documentation for details on the settings:
https://sonicontrol.fhstp.ac.at/documentation/

Settings Activity



Rules & Detections Activity



update notification

start

Main Activity

show settings
close
stop
end
show rules & detections
delete entry
close
start
pause / stop
read (get all entries)
write (delete entry)

scan & detect...

no
Message Detected?
yes

store hi-pass file to disk
recognize
compute detection scores (see recognition diagram)
max → get technology
get location

Outputs
- technology, e.g. „lisnr“
- lon/lat
- high-pass filtered wav file

no
Block on each detection?
yes
yes
no
New Sound?
yes
block until user makes a decision
no
Should block? (in location?)
yes
always

{
Signals: [
list of detection items: {
lon/lat
address
technology
technology ID
last detection
blocking status
detection counter
amplitude
}
],
Signal_History: [list of detection items...],
Imported_Signals: [list of detection items...]
}

Store on external SD card. If the user does not provide the permission for it, then we operate without JSON file, i.e. we ask on each detection to block or not.

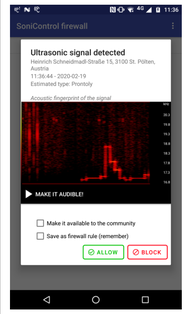
JSON

SoniControl Server

send to server
playback (pitch shifted)
make available to communtity: checked
make audible

read (search for entry at location of recognized type)
write „last detected at [date/time] (location must be checked before))“

Alert



allow
block
block / allow

no
Use microphone for blocking?
yes
no
Get Mic Access?
yes
jam for N minutes
update notification
block mic for N minutes
get location
still in location?
yes
no
or until user made a decision in „Alert“

write (new entry in `Signal_history` and optionally as firewall rule in `Signal` array if „save as firewall rule“ is checked

SoniControl Detection Process

Detection Parameters

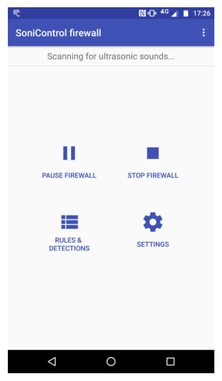
Buffer sizes:
- `bufferSize`=50; %ms
- `backgroundBufferSize` = 10; %sec
- `medianBufferSize` = 2.5; %sec, recommended values between 1s and 2.5 seconds

Detector parameters
- `cutoffFrequency` = 16800; %lower limit for prontoly!
- `decisionThreshold` = 0.5; %this is for Kullback Leibler Divergence
- `percentSilenceAfterDetection = 10;` % unit=percent of `medianBufferSize`
- `extendedDiagnostics = 0`; % Delays the alert until the detected message has ended. This produces detections which are more likely to contain the entire message. Thereby, we can visualize the entire message and can enable more enhanced diagnostics of the detected message. Valid values: 0 or 1.

Recognition parameters
- `specs.nearby.nBands`=64;
- `specs.nearby.bw=(20000-18500)./specs.nearby.nB`ands/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

pause / stop

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 fulfills detection condition `detectionAfterMedian` = 0; %becomes one only if more than the half of the frames in the medianBuffer fulfill the detection condition

Loop (until we have a detection) OR paused

`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

append samples of current buffer to bufferHistory

save current buffer in `bufferHistory`

get median

compute median of `backgroundModelBuffer`

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

if `backgroundDist` > `decisionThreshold`

no

yes

`detection` ← 0

`detection` ← 1

update `medianBuffer`: `medianBuffer` ← `detection`

push

Is `backgroundModel Buffer` and `medianBuffer` full?

no

yes

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

get median from `medianBuffer`

if `extendedDiagnostics` == 0?

no

yes

wait until message ends, i.e. median buffer switches to 0 and stays mostly 0 after a detection. „Mostly 0" means that at least 75% of the duration defined by `percentSilenceAfterDetection` the `medianBuffer` is zero (this means, a small amout of outliers (25%) are tolerated

median is 1?

no

yes

update `backgroundBuffer` with `bufferFFT_HiPass_Norm`

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

to remove potential foreground sound in the background model)

update

update

Do signal recognition

`detected technology`

store hi-pass file to disk

get location

SoniControl Recognition Process