Final Documentation ComFrame

David Krebs, TU Munich David Zhang, TU Munich

October 15, 2014

Contents

1 Introduction

1.1 Background

In the context of the "Android Praktikum" our two-man group, consisting of computer science students at the Technische Universität München, has taken up the challenge of the project named ComFrame. The project follows the idea that usability of apps should not be impaired by user-permission requirements popping up every time a new communication channel is build up between devices, in contrast to Bluetooth or Wi-Fi. Thus we need a new method for inter-device communication, which ComFrame provides, by transmitting data via sound. Of course this sound should not disturb users and is designed to be inaudible. This was achievable to a certain extent, as the sound itself is transmitted at an in generally inaudible frequency, but some noise is produced as a byproduct of the modulation in use. The main result of this project is a library, to be used by other developers as an API, and we also highly encourage interested people to further improve the framework.

1.2 Contributor

Our group consists of David Krebs and David Zhang, who are both studying computer science at TUM in Garching and are at the time of this project in their 4th Semester of Bachelor. Both group members went to school in Germany, got their entrance requirement for higher education at the same time and started their studies at the TUM right away.

1.3 Main Idea

The main idea of ComFrame is to build a communication framework, which transmits data via inaudible sound. In concrete, we want to construct an intercommunication interface between two android devices, without relying on Bluetooth or Wi-Fi. This is useful, as it does not require any user permissions to utilize the microphone or loudspeaker. This project is also designed to be Open Source and hopefully further improved by other interested developers. Additional to the framework we offered to implement a sample application, relying on the ComFrame to communicate, to demonstrate the usage of our framework, implemented as a library. This resulted in the game app ”TicTacToe”, which can be played in a multiplayer mode by two devices in close vicinity. We also offer a Java Doc and a developer guide.

2 Project Flow

Nils first presented the project's main idea. We decided to take this up, because it sounds interesting, to create a new way of communication between android devices. We did so despite our lacking knowledge in DSP (digital signal processing). At the start we had to get broader and deeper knowledge on the different modulation methods, as there are three general possibilities of modulating data. Shifting the amplitude was not a possibility, which is obvious, because of the variable range between the two devices and frequency shift keying was also excluded, since our bandwidth was limited to inaudible sound. We decided on PSK (phase-shift keying), because the other methods did not offer the necessary qualities we looked for. Then having chosen a general modulation method, our group used a specially developed app to test and analyze some transmitted sound. We started with simple sinus waves of frequencies in the range of 18kHz – 20kHz and made those more complex over time. This enabled us to have a look at the phases of the signal we received and led us to the conclusion that PSK was not practical, because of the absolute frequency being not reliable enough. So we chose to use DPSK (differential phase-shift keying) instead, as it only uses the phase difference between two symbols. The next problem we encountered was, that we could not rely solely on the amplitude to discern, if our carrier wave is present or not. This ambiguity had several causes, e.g. a noise spike could happen, just by physically "scratching" parts of the mobile device. Though the most unsettling reason was, that the signal lost amplitude over time, despite not having changed the volume of the speaker (from the sending device), which seems to be a feature of Android to filter frequencies that have been too loud a certain time span (about have a second, but differs between different devices). Our first attempt to solve the problem of sudden noise spikes was, by counting a certain number of symbols that pass our magnitude threshold (the volume is high enough), in a certain period of time. We realized after thorough testing, that this was still not enough. After some thinking we tackled the problem in a new way, by first "counting symbols", which is based on the same thought, as the previous attempt, that there is data on our communication channel only if a certain amount of symbols pass the threshold. Though this itself was not enough to decide, whether there is data or just noise, so we also added a starting sequence followed by a length field. The concrete algorithm is explained in the project description part. Another problem was, the "noise" that could be heard, while sending some data. We realized it was produced by the phase-shifts and was hearable despite the carrier wave being at an inaudible frequency. This was a problem that we could not solve despite trying several things, like using a "ramp" (simply said, just lowering the sound volume around the phase shifts) or other things. We tested other Apps that are also trying to send data via inaudible sound and concluded that the same noise was hearable there as well. A major problem we also had, is the receiver not knowing when the exact position of a symbol-start is. This synchronization problem took us quite some time to solve and still poses a problem because of other causes, that will be mentioned later. Originally we only used two arrays of same size (the size can be seen "equivalent" to the symbol time, because the sample rate is fixed) to send a symbol and to receive those, but because of the missing symbol synchronization between the two devices, we implemented an algorithm to detect the "best" way to align our receiving buffer to the measured (by the microphone) signal. As before mentioned we had a new upcoming problem, namely more de-synchronization. This occurred after having received some (variable) amount of data, in specific, the two testing devices lose their symbol synchronization, which leads to unusable phase differences. Although we made several attempts at solving this, we could not fully get rid of it. We suspect the main cause to be directly related to the hardware of one of the devices we used for testing. Furthermore we did some testing and decided based on empirical research that we profit more from using SDPSK (symmetric differential phase-shift keying) instead of DPSK. This profit is meant in the sense of a lower error rate. It is also to be mentioned, that the process of gaining the background knowledge, required to not only understand how DSP works, but to also apply it onto our android library, was quite time intensive, because of the abundance of information presented in circuits or circuit terminology and as well the lack of software related descriptions.

3 Project Description

As this project was originally proposed as an open source framework (though we added a fully implemented app) to be used and further improved by other developers, we have designed an easy to use interface to integrate it into another apps. We also paid attention to make it "developer friendly" to encourage others to further improve it. Mainly by dividing it into five parts:

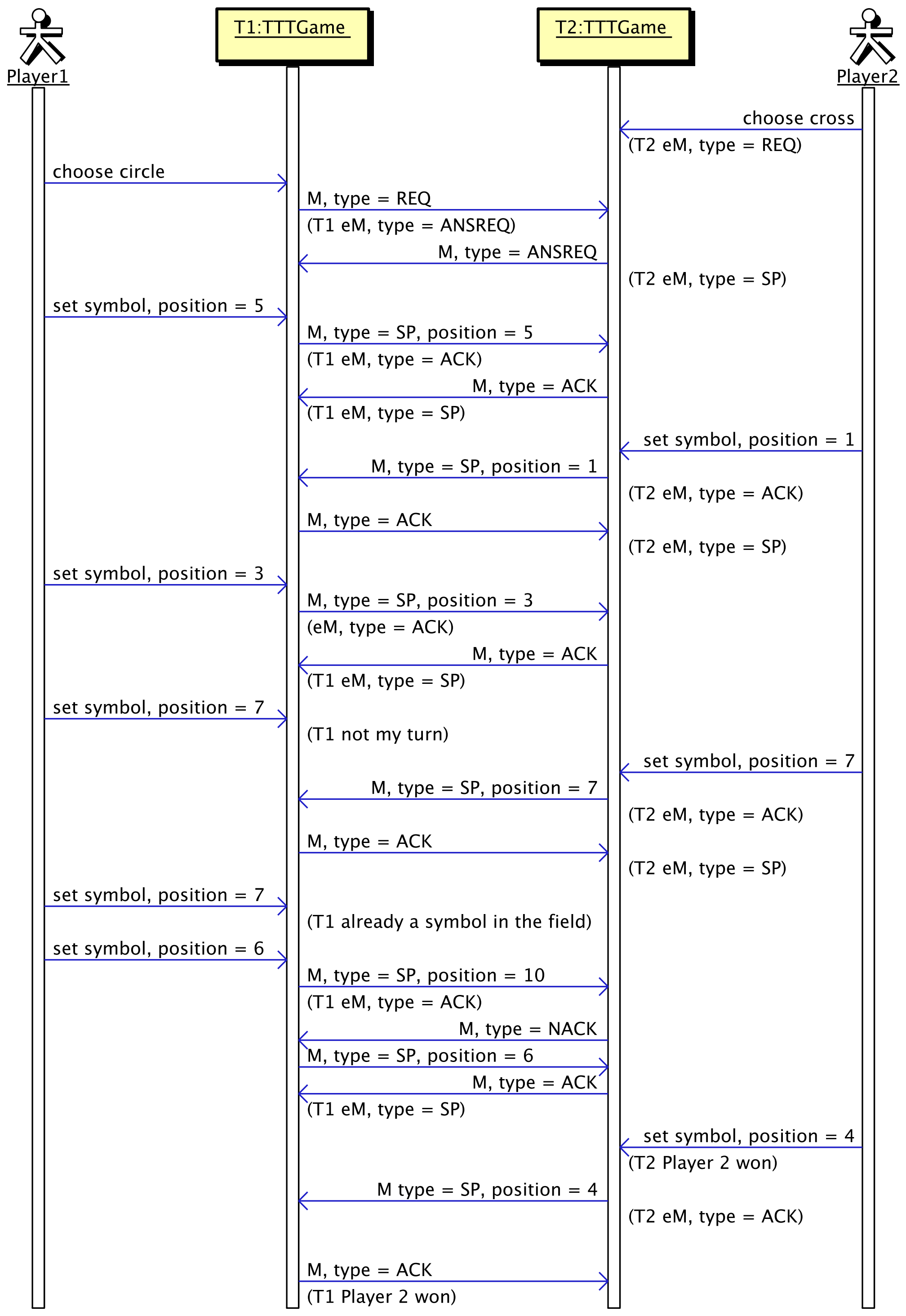
- The interface of the framework, which are basically the ComFrame-Sender and ComFrameReceiver classes.
- The modulation part, where we are currently using SDPSK.
- Channel coding, where different hamming codes are used.
- The "math" that is needed to process signals.
- (Some basic functions needed in several classes that are not directly provided by Java.)

The 1. Part offers, as mentioned, an interface for developers. For a specific user case you should take a look at the developer guide, which is also included in this project. The methods that are offered are simply sending and receiving messages. The next part consists of a class for modulation and one for demodulation. As aforementioned we use SDPSK to encode and decode our data into the carrier wave. This modulation process works by adding a phase of -$\frac{\pi}{2}$ or $+\frac{\pi}{2}$ for a binary 0 or respectively a 1. (PSK in general works by adding and identifying phase shifts to the signal. The difference of DPSK to PSK is, that we interpret the data differently. Instead of realizing the symbols with instantaneous phases, we use the phase difference. The phase shifts for DPSK are generally done by adding 180° for a 1 and 0° for a 0. SDPSK changes these shifts into a symmetric variant, by adding 90° instead of 180° and -90° instead of 0.) Our carrier wave is a single sinusoid function with the frequency 18.433kHz, which we sample at a rate of 44.1kHz. The frequency is chosen as it is in general inaudible and the sample rate needs to be at least double of the sound frequency, according to Nyquist, which leads us to the given number. For demodulating the signal we first have to sense our carrier signal, which is done in several steps as indicated in the paragraph about the project flow. First we wait, until we have enough "raw data" (provided by the microphone), then we check if it is possible to align our symbol buffers in such a way onto the signal, so that we get a desirable phase difference. We do this by choosing different “start points” in the first buffer array (of the size of a symbol) and calculate the maximum “error” each of the following symbols has, where the “error” is the difference between a phase shift of $\pm\frac{\pi}{2}$ and the calculated phase shift. Afterwards we choose the best “start point” by taking the one with the minimal (maximum) error. These so called “start points” are simply fixed equidistant points in the first symbol buffer. Now we check if most of these symbols pass our magnitude threshold. If they do, we can compare the decoded data to our predefined start sequence. And finally, if this yields true, we judge it to be some data send by another device and synchronize with it by skipping some “raw data” at the beginning, that we deem not to be part of our signal. When we are done with this, we decode the length field next, which comes right after the start sequence and also has a predefined size. The length field encodes the amount of data following, in bytes. After getting the length, we demodulate each symbol in the sequence it arrives, until the length is reached. Our project also offers an option for channel coding, in concrete we offer Hamming(7,4) and Hamming(15,11). The usage is optional and can be easily enabled, for more information, refer to the developer guide. The math part currently contains only the Goertzel class, which implements, as the name suggest, the Goertzel-Algorithm. It yields the same result as the DFT (discrete Fourier transform), though for one (a-priori decided) “point” only. This only works for a real value input though, which is not a problem in our case. In comparison to other FFT implementations this one fits our needs best and provides much more efficiency. We decided for this to be in a separate package, to emphasis the logical separation between the modulation and the algorithm used to analyze the signal. It could theoretically be replaced by any other algorithm that may fit the needs of a future developer (maybe some special hardware based library). The last part just consists of some basic functions, that are necessary in general, but not directly offered by the Java or Android API. The game app "TicTacToe" uses our library for inter-device communication. This way, we can enable two players to play at the same time against each other. The graphical interface is kept simple, as it is only designed as an example of how and where to use ComFrame. The game it self follows the same rules as the well known game with the same name. It starts by offering the possibility to choose either the cross or a circle as the symbol to play with. The circle starts by default and also sends a request to play with, to the other device, to start a game session, while the cross waits for an incoming request. This is of course done in the background and transparent to the player(s). The communication part and the game logic are implemented in the class TTT. Apart from that we use two different Activities and a custom View to enable to enable interaction between the user and the device. The first Activity is the MainActivity, which provides the starting panel for the user. The second activity only includes our custom View that is basically the playing field. We devised a communication protocol that consists of 5 different message types:

1. Request (REQ): Sender requests to start a game session, as stated before.
2. Answer (to) Request (ANSREQ): Answers an incoming request message from the other device.
3. Acknowledgement (ACK): Acknowledges an successfully received message containing a symbol (cross or circle) position.
4. Not acknowledged (NACK): Request the other party to resend the previous message, because of an erroneous received data.
5. Symbol position (SP): Sender tells the other party, where the player set his or her symbol at, provided it was his or her turn.

The incoming messages (provided by ComFrameReceiver) are all handled depending on the currently expected message type. The message type that is expected next, depends on the last send or received message. The concrete message contents were designed to withstand bit errors to a certain extent, meaning that it did not matter if e.g. a request has 2 bit errors, since it mostly resembles the expected message. In the following the sequence diagram depicts a possible course: M: Message eM: The message that will be expected next. type: This is an attribute, having the type of the send or expected Message. position: A number from 1 to 9 with a bijective mapping to all (row, column)-tuples, with 3 rows and 3 columns.

1	2	3
4	5	6
7	8	9


Player1
T1:TTTGame
T2:TTTGame
Player2
choose cross
(T2 eM, type = REQ)
choose circle
M, type = REQ
(T1 eM, type = ANSREQ)
M, type = ANSREQ
(T2 eM, type = SP)
set symbol, position = 5
M, type = SP, position = 5
(T1 eM, type = ACK)
M, type = ACK
(T1 eM, type = SP)
set symbol, position = 1
M, type = SP, position = 1
(T2 eM, type = ACK)
M, type = ACK
(T2 eM, type = SP)
set symbol, position = 3
M, type = SP, position = 3
(eM, type = ACK)
M, type = ACK
(T1 eM, type = SP)
set symbol, position = 7
(T1 not my turn)
set symbol, position = 7
M, type = SP, position = 7
(T2 eM, type = ACK)
M, type = ACK
(T2 eM, type = SP)
set symbol, position = 7
(T1 already a symbol in the field)
set symbol, position = 6
M, type = SP, position = 10
(T1 eM, type = ACK)
M, type = NACK
M, type = SP, position = 6
M, type = ACK
(T1 eM, type = SP)
set symbol, position = 4
(T2 Player 2 won)
M type = SP, position = 4
(T2 eM, type = ACK)
M, type = ACK
(T1 Player 2 won)

List of (implemented) functions and some comments:

- ComFrame API: The basic functions of sending data from one device to another are of course implemented. The question of, whether we also met our “quality goal” is not easily answered. In the beginning we stated that we want to optimize the data and error rate, but these two criteria are conflicting with each other and for that reason there had to be compromises made. As there are certain hardware dependencies to the error rate, we cannot offer a fixed number, although we tried to keep it as high as possible. This resulted in a lower data rate, which is currently at around 172bit/s. To objectively asses “how much of the goal” we reached is complicated, as there are several factors to be considered. For instance the amount of time that was estimated for each student (165h/student) was by far too less for this project. The actual time necessary to produce the result we achieved was by far more then that. Another factor that has to be considered is the fact that we had no prior experience at applying DSP into software. So the data rate can probably not compete with commercial versions of similar applications. Though it should be mentioned that, after testing one of these application (using SDK of NearBytes), we could find similar weaknesses, like the afore mentioned “noise” while sending data or as well the reliability problem of recognizing signals correctly. Apart from this, our API also offers a feature that was optional. Namely the channel coding, implemented as Hamming Code(7,4) and Hamming Code(15,11).

- Simple Sample App: Our original goal was to only implement a very simple and basic App to provide an example as to how to use our API. This idea was extended to the implementation of a well know game and the development of an underlying protocol for interaction between two devices.

- Javadoc & Developer Guide: This was planned from the beginning and is also included.

As a reference for a test plan you can either use the game app or if it is necessary to test the ComFrame library itself, it can be done by following the developer guide to implement a testing app. The game app is quite intuitive to use and a possible test case can be found in the sequence diagram.