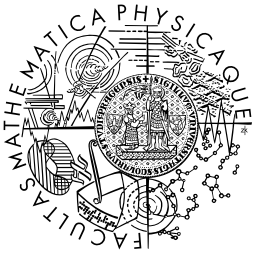

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Hrach

Frequency Spectrum Monitoring System

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis:	Mgr. David Klusáček, Ph.D.
Study programme:	Informatics
Study branch:	General Computer Science

Prague 2016

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank to my supervisor for the great signal processing classes showing us marvels of mathematics and to members of brmlab, the Prague hackerspace, for collaboration on various radio and other projects and providing valuable feedback on software presented in this thesis.

Title: Frequency Spectrum Monitoring System

Author: Jan Hrach

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. David Klusáček, Ph.D., Institute of Formal and Applied Linguistics

Abstract: We have created several programs to be used in the field of Software Defined Radio technology: (i) A channelizer, a tool that splits wideband signal into multiple narrowband channels that can be further processed, (ii) A SDR client, an interactive GUI tool that allows exploration of signals, (iii) A scanner. Our channelizer is several times faster than a popular GnuRadio implementation and our SDR software features a network-transparent architecture with one server and multiple concurrent clients.

Keywords: SDR channelizer filterbank DSP GnuRadio

Contents

Introduction

The SDR concept enables us to create receivers and transmitters of various signals using single universal hardware peripherial by simply replacing the control software. Thanks to the falling prices of such universal hardware devices, wide availability of powerful computers and inherent ability to share the software among developers, we have seen a large boom of SDR usage in recent years.

This work aims to create tools that aid the development of SDR applications and can be used for learning basic principles of the SDR technology and radio reconnaissance.

In Chapter 1 we give general introduction to the SDR technology, motivation and available hardware. In Chapters 1.3 and 1.4 we describe two popular programs that can be used with SDR — GnuRadio and GQRX — with emphasis on identifying their advantages and drawbacks.

In Chapters 2–4 we present three programs we have developed to address the identified drawbacks.

- Chapter 2 describes a spectrum channelizer, a tool that splits wideband signal into several narrowband channels that can be further processed.
- Chapter 3 describes an interactive SDR client, a graphical program where the user can see available transmitters and decide to receive some of them.
- Chapter 4 describes a scanner, a program tuning to various frequencies, looking for new signals and saving them for later analysis.

1. The Software Defined Radio

1.1 Motivation

The "classical" radio implements the signal processing and demodulation in hardware using features like tuned filters, mixers, etc. This makes any reconfiguration difficult, usually requiring major changes in the physical circuitry. Under the term "software defined radio" (from now on occasionally only SDR) we understand the approach of sampling the received signal by an analog to digital converter (ADC) as early as possible and then implementing all the signal processing on a general purpose computer. Similar approach also applies to transmission, when the signal is generated in software and then transmitted using a digital to analog converter (DAC). By general purpose computer we also mean signal processing accelerated by reconfigurable hardware like, for example, a field-programmable gate array.

The advantages of this approach include for example:

- Single hardware can be reused for the reception of various modulations and protocols by only changing the software.
- Usual software development tools like debuggers and version control systems apply.
- Both the software and the sampled signals can be stored and distributed over the computer networks.

The "ideal" SDR would consist of an antenna, an amplifier, a low-pass filter and an ADC. However, to meet the Nyquist criterion, one would need an ADC with a sample rate of two times the received signal frequency (and some safety margin for the low-pass filter transition band). This is infeasible for signals in the VHF and higher bands.

We have two other options here. Either to use a bandpass filter, deliberately violating the Nyquist criterion and sample the alias, or mix the signal with local oscillator to convert it to some lower intermediate frequency. We use the former approach when discussing a channel selector in Chapter 2. Most general-purpose SDRs (for example rtl-sdr, Airspy and bladeRF) use the latter approach.

1.2 SDR hardware

Some popular commercially available receivers are summarized in Table 1.1.

Device	**Samplerate**	**Resolution**	**Frequency range**	**Cost**
rtl-sdr	2.4 MHz	8 b	24–1800 MHz	$8
Airspy	10 MHz	12 b	24–1800 MHz	$220
bladeRF[a]	40 MHz	12 b	300–3800 MHz	$420

Notes: [a] Can transmit and has user-configurable FPGA

Table 1.1: Popular SDR hardware

1.3 GnuRadio

[GnuRadio] is a software library that provides signal processing primitives in so-called blocks. Each blocks does one operation with the signal, for example multiplication with a constant, Fourier transform or FIR filtering, and blocks can be connected into flowgraphs.

Unfortunately, the architecture does not scale well when running many blocks in parallel. We hit this bottleneck with GnuRadio PFB channelizer and decided to implement our own channelizer, which we discuss in Chapter 2.

1.4 Gqrx

[Gqrx] is a graphical software built upon the GnuRadio library. It is to be run on a computer which has a SDR hardware connected. It prompts the user for a frequency, tunes the SDR to that frequency and then shows the power spectrum of the sampled signal. The user can choose to demodulate the signal using one of several demodulators; as of version 2.5, FM, AM, sideband and AFSK-1200 (digital data used in APRS networks) are available.

We have been using Gqrx as the main debugging tool during development of various radio projects and we have created a list of features that we consider missing.

Network transparency. The only way to operate a remote SDR in Gqrx is to send all the data from the SDR over the network. This usually means tens or even hundreds of megabits per second. It works over LAN, but cannot be used over the Internet.

Our approach is to process the data remotely and send to the client only what is really needed — spectrum measurements and filtered narrowband channels. With this, the required bandwidth can be only few Mb/s, depending on the number and bandwidth of the channels.

Multiple demodulators running at once. SDRs can sample the spectrum many megahertz wide (see Table 1.1), therefore several signals of interest may be received simultaneously. Why one cannot add multiple demodulators in Gqrx?

History browsing. A short burst of signal can appear and the user might be interested in it. However, it disappears before the user can tune to it.

Current computers have many gigabytes of memory. It is therefore no problem to store several minutes of everything that the has SDR received in a ring buffer. Convenient GUI can then allow to dump the entire buffer to disk for offline analysis or just pick a region limited by frequency and time.

Pluggable demodulators. We just want to provide a new program and have the data piped to it.

Advanced squelch. Squelch is a function that detects signal presence and disables the output when no signal is present (so the user is not annoyed by noise, or, alternatively, system resources are not wasted by trying to decode empty channels). Usually, this is implemented by measuring the absolute signal level and setting the threshold. Unfortunately, the absolute signal level can vary wildly because of changing atmospheric and interference conditions or when the antenna is physically moved. Measuring squelch level with reference to the surrounding channels proved to be much more stable in our implementation.

Additionally, the squelch should employ some averaging to avoid false positives on short bursts of noise. However, this could mean that the beginning of the transmission could be cut until the moving average detects the signal. So the squelch function should hold a small buffer and replay it when it triggers.

Histogram. Setting the correct gain of the input amplifier can be tricky, especially with low-end SDRs that have low dynamic range and are prone to saturation. Plotting a real-time histogram of samples proved to be an extremely useful help.

Automatic frequency correction. Frequency of the signal may drift due to instability of transmitter and receiver oscillators or due to factors like Doppler shift, which is a concern when receiving signals from fast-moving objects like satellites. Additionally, the user may not know the frequency accurately. The software should automatically determine the exact frequency of the signal and make precise tuning.

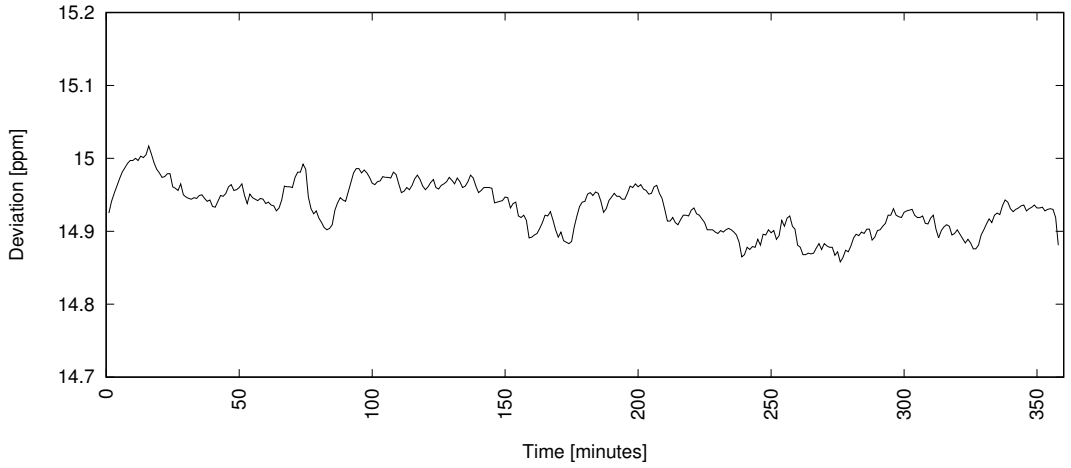

Figure 1.1: Frequency drift of rtl-sdr receiving GSM900 signal for six hours. Data were obtained by running [kalibrate-rtl] and `rtl_sdr` in a loop. The standard deviation of magnitude 0.02 ppm is not shown.

Scanner. The SDR could be run unattended, looking for rare signals (ranging from IOT sensors that only report their status once per hour over direct communications used during emergency to radioastronomical measurements looking for ionized trails of meteors), saving them and generating summary reports.

We have searched for software alternatives and at least some of the aforementioned problems seem to be present in all available SDR programs (plus some more, like some of them being closed-source or Windows-only). Then we considered implementing these features into Gqrx, but finally we decided to design our own program from scratch. This is described in Chapters 3 and 4.

2. The spectrum channelizer

Many modern networks, for example GSM, TETRA, Tetrapol, or even FM radio broadcast, use frequency division multiplex with a lot of transmitters. This chapter describes how to efficiently receive multiple transmitters at once. The theoretical fundamentals of the algorithm were studied from Chapters 6 and 9 of the book [Harris, 2004] and the implementation was discussed in the [Klusáček, David, 2015] course. The channel oversampling was inspired by the channel scheduler in [gr::filter::pfb_channelizer_ccf].

2.1 The channel selector

Consider that a frequency division multiplex has been received. The FDM consists of several equally spaced channels and without the loss of generality one of the channels is centered at frequency 0 (if this cannot be achieved, multiply the signal by complex heterodyne so that this holds). The spectrum of the signal looks like this:

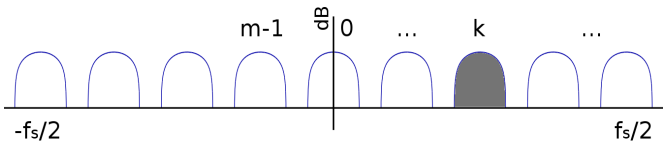

Figure 2.1: Spectrum of several channels of a frequency division multiplex.

Suppose we are interested in channel k. We apply complex rotator to translate it to frequency 0, filter it with a low-pass filter and downsample it to the required target rate.

Let x denote the complex input signal (and x_j the j-th sample), m the total number of channels, h the real FIR filter (and h_j its j-th coefficient), N the order of the filter h and d the downsample ratio. We have:

$$\text{rotated}(n) = x_n e^{-2\pi i \frac{k}{m} n}$$

$$\text{filtered}(n) = \sum_{j=0}^{N-1} \text{rotated}(n-j) h_j$$

$$\text{filtered}(n) = \sum_{j=0}^{N-1} x_{n-j} e^{-2\pi i \frac{k}{m}(n-j)} h_j \tag{2.1}$$

$$\text{downsampled}(n) = \text{filtered}(nd) = \sum_{j=0}^{N-1} x_{nd-j} e^{-2\pi i \frac{k}{m}(nd-j)} h_j \tag{2.2}$$

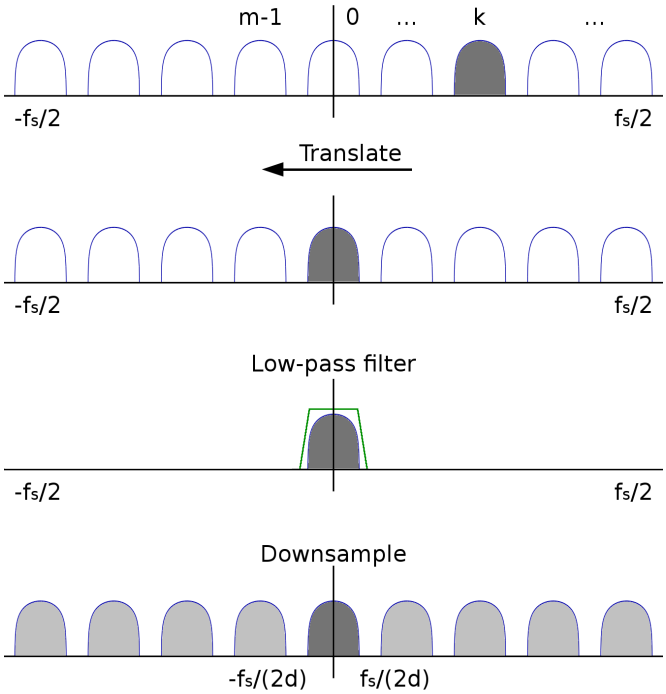

Figure 2.2: Applying complex rotator, a low-pass filter and downsampling a FDM

The complexity of the algorithm for one output sample is $2d$ complex multiplications for the input rotation (one for actually rotating the sample and one for updating the current phase shift) and $2N$ real multiply-accumulate operations for filter evaluation.

Let us rewrite the equation (2.1) as

$$\begin{aligned} \text{filtered}(n) &= \sum_{j=0}^{N-1} x_{n-j} e^{-2\pi i \frac{k}{m} n} e^{2\pi i \frac{k}{m} j} h_j = \\ &= e^{-2\pi i \frac{k}{m} n} \sum_{j=0}^{N-1} x_{n-j} (e^{2\pi i \frac{k}{m} j} h_j). \end{aligned} \tag{2.3}$$

We now observe that the term $e^{2\pi i \frac{k}{m} j} h_j$ is not dependent on the input signal. We can precompute a rotated filter h' defined by

$$h'_j = e^{2\pi i \frac{k}{m} j} h_j$$

and from (2.2) we have

$$\text{downsampled}(n) = e^{-2\pi i \frac{k}{m} nd} \sum_{j=0}^{N-1} x_{nd-j} h'_j,$$

hence we save repeated complex multiplications by $e^{2\pi i \frac{k}{m} j}$ when evaluating the sum at the cost of evaluating complex FIR filter instead of real.[1] Systems with

[1] Applying real FIR to a complex signal is the same as applying real FIR to two real signals, where one is $\text{Re}(x)$ and the second $\text{Im}(x)$. Applying complex FIR to a complex signal requires complex multiplication, which needs 6 floating point operations.

limited memory bandwidth may additionally benefit from the fact that we don't need to store the entire rotated baseband. For very long filters and small downsample ratios, some variant of FFT-assisted convolution like overlap-add may be considered.

The resulting complexity for one output sample is N complex multiply-accumulate operations for filter evaluation and 2 complex multiplications for the final rotator. Whether the first or the second algorithm is faster will depend on actual parameters (FIR order and decimation) and hardware details of the target platform (e.g. speed of vectorized multiply-accumulate instructions). An advanced implementation may measure the speed of different implementations for various parameters and then pick the best one at runtime. Currently we don't think channel selection is a bottleneck in applications where we use it, so we leave this as a possible future improvement.

Intuitively, one can imagine the optimized algorithm as transforming the low-pass filter to a band-pass filter and then deliberately violating the Nyquist criterion and using the spectral alias that appeared.

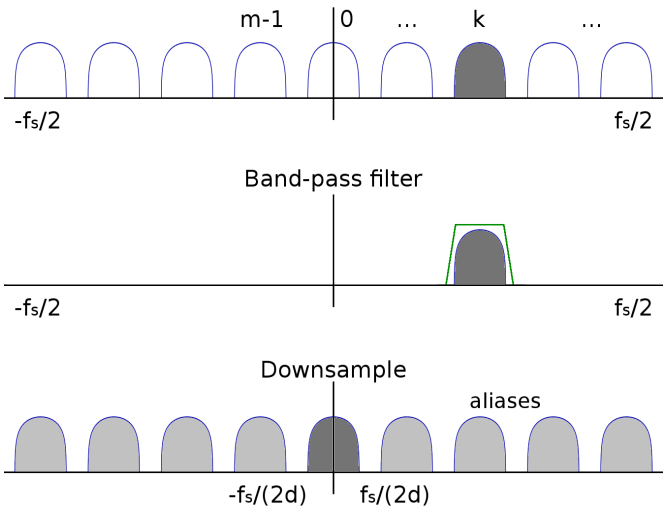

Figure 2.3: Applying a band-pass filter and downsampling a FDM

The first algorithm is implemented in file `kukuruku/scanner/xlater.c` and the second in `kukuruku/server/xlate_worker.c`.

2.2 Computing multiple channels at once

Receiving multiple channels from the same FDM using multiple channel selectors might be computationally infeasible. The channelizer can compute all FDM channels with lower computational demands.

First, assume that $m = d$, i.e., we want to compute non-overlapping channels. Recall again the equation (2.1), but let us rewrite the filter as a real matrix g of size $N/m \times m$ (if the order of the filter is not divisible by d, pad it with zeroes).

$$g_{q,p} = h_{pm+q}$$

$$g = \begin{pmatrix} h_0 & h_m & \cdots & h_{N-m} \\ h_1 & h_{m+1} & \cdots & h_{N-m+1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{m-1} & h_{2m-1} & \cdots & h_{N-1} \end{pmatrix}$$

Hence after rewriting (2.2) with the filter "matrix" by evaluating the filter by rows and columns and substituting m for d,

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} \sum_{p=0}^{\frac{N}{m}-1} g_{q,p} x_{nm-j} e^{-2\pi i \frac{k}{m}(nm-j)},$$

where $j = pm + q$, so we have

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} \sum_{p=0}^{\frac{N}{m}-1} g_{q,p} x_{nm-pm-q} e^{-2\pi i \frac{k}{m}(nm-pm-q)}.$$

Let us define $w_{n,q} = \sum_{p=0}^{\frac{N}{m}-1} g_{q,p} x_{nm-pm-q}$, so we can write

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} w_{n,q} e^{-2\pi i \frac{k}{m}(nm-pm-q)}. \tag{2.4}$$

We notice that k, n and p are integers and $e^{-2\pi i u}$ is 1 for integer u, so $e^{-2\pi i k n}$ and $e^{-2\pi i k p}$ evaluate to 1 and we can write

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} w_{n,q} e^{2\pi i \frac{k}{m} q}.$$

Recall and compare this to the definition of the inverse discrete Fourier transform,

$$\text{IDFT}(x)_k = \sum_{n=0}^{N-1} x_n e^{2\pi i k n / N}.$$

So we can get a sample for every channel k by evaluating partial filters $\sum_{p=0}^{N/m-1} g(q,p) x(m(n-p)-q)$ for every q and then Fourier-transforming the resulting vector.

The data flow in the algorithm can be illustrated in the following figure.

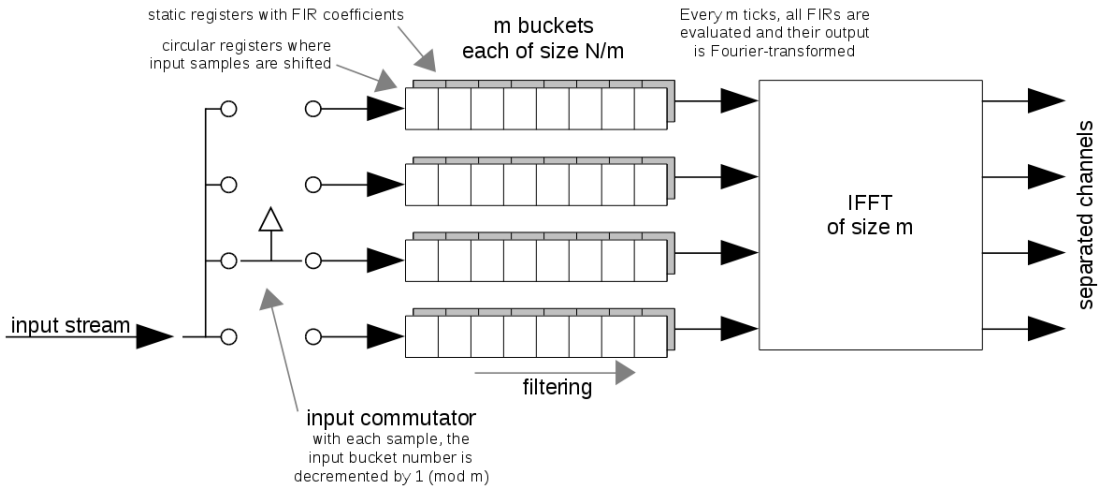

Figure 2.4: A spectrum channelizer

The complexity for m output samples (one sample of each channel) is $2N$ real multiply-accumulate operations for filter evaluation and one FFT of size m, which takes, depending on implementation, about $5N \log N$ floating-point operations.

2.3 Oversampling

We have assumed $d = m$, i.e., the resulting channels do not overlap. This is not always convenient, for example the Tetra network has channel spacing 25 kHz and symbol rate 18 kHz and the Tetrapol network has channel spacing 12.5 kHz and symbol rate 8 kHz and the demodulator usually requires 2 samples per symbol (see [Nutaq] for further discussion). It would be therefore convenient if we could have m such that channels are spaced by 25 kHz or 12.5 kHz and d such that each channel has 18 kHz $\cdot$2 samples / symbol = 36 kHz (Tetra) or 16 kHz (Tetrapol) samplerate.

We call this oversampling and we will show that d and n can be arbitrarily chosen positive integers. With distinct d and m, we have from (2.4):

$$w'_{n,q} = \sum_{p=0}^{\frac{N}{m}-1} g_{q,p} x_{nd-pm-q},$$

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} w'_{n,q} e^{-2\pi i \frac{k}{m}(nd-pm-q)}$$

Again noticing that k and p are integers, $e^{2\pi i \frac{k}{m} pm} = 1$, so

$$\text{downsampled}(n) = \sum_{q=0}^{m-1} w'_{n,q} e^{-2\pi i \frac{k}{m}(nd-q)}.$$

As the sum is just running q from 0 to $m-1$, we can rearrange the order of summands, so let $q' = (nd - q) \bmod m$. Therefore

$$\text{downsampled}(n) = \sum_{q'=0}^{m-1} w'_{n,((nd-q') \bmod m)} e^{-2\pi i \frac{k}{m} q'} \tag{2.5}$$

and we have a (forward) discrete Fourier transform again.

2.4 FCL: the implementation

A C++ implementation of the algorithm is available in GnuRadio. Unfortunately its performance seemed to be surprisingly poor. We hypothesize it is caused by frequent thread switching and non-optimal layout of the filters in memory, but we haven't investigated the problem any further.

We have decided to implement the algorithm from scratch to see if a standalone implementation can be faster.

The "buckets" (partitioned FIR filters) are implemented as ring buffers. To avoid modular wraparound, we precompute the bucket filter for every possible position of the ring buffer pointer and then use plain multiply-accumulate. With this, no modulo operations are needed, the compiler can easily vectorize the loop and memory prefetch does not need to care about the wraparound. Of course all these precomputed filters take some memory, but with the usual FIR order of about 2000 and 100 buckets, this means $100 \cdot (2000/100)^2 \cdot \text{sizeof(float)} = 160\,\text{kB}$, which fits into cache.

We use multithreading; if t threads are running, each thread computes every t-th output sample. The main thread reads input data in chunks of about one megabyte (set in `finetune.h`) and synchronizes these worker threads using mutexes and conditional variables.

To allow full loop unrolling, we compile multiple .so modules, each designed for one bucket size, and one generic module for any bucket size. The right .so is determined and loaded at runtime. However, this does not seem to contribute any performance gain after all optimizations were implemented.

Our channelizer can read samples directly in `float32`, `uint8_t` and `int16_t` formats, so the input stream can be directly piped into it from low-level SDR utility like `rtl_sdr`, which further reduces resources demand.

Before channelization, the signal passes through a rotator, which allows to impose a frequency shift. This can be used for runtime compensation of SDR clock drift.

Finally, we have added a TCP server into the program. It can be used for runtime reconfiguration - setting the frequency correction, determining power spectrum of the input signal and selecting which channels we want to output.

2.4.1 Usage

The program is available in the Attachment 1. Build can be achieved by simply running `make`. The dependencies are the FFTW library and VOLK.

The program reads samples from stdin and writes them to a file (which could be a named pipe). The output channels are interleaved, i.e., the first sample of each channels is outputted, then second sample from each channel etc. When run without parameters, it prints usage information. You need to somehow generate

the FIR filter you want to use. The FCL itself runs the program specified by the `-f` parameter and expects it to output real numbers, one per line. We provide a small wrapper around the GnuRadio Firdes module which can be used as this program. It is called fir.py and again, if run without parameters, it prints simple usage information. For example,

```
fcl> ./fir.py 2e6 40e3 15e3
```

creates a low-pass filter that when applied to a 2MS/s signal has 40kHz passband and 15 kHz transition band (as we are applying a real filter to complex signal, it is symmetric, so the actual passband would be from -40 kHz to +40 kHz).

The most important parameters are `-n` and `-s` specifying variables m and d from Equation (2.5), `-o` specifying the output file and `-c` specifying the list of comma-separated channels we want to output. The input format can be chosen by the `-i` parameter, default is complex float. The following command reads 2 MHz of FM broadcast from rtl-sdr, channelizes it to 20 channels with 200 kHz bandwidth each and outputs channels 3 and 8.

```
rtl_sdr -f 90e6 -s 2e6 - | ./fcl -n 20 -s 10 \
 -f "./fir.py 2e6 75e3 10e3" -c 3,8 -i U8 -o /tmp/myout.ch
```

FCL offers a simple telnet interface on localhost:3333 (host and port can be changed with `-b` and `-p` parameters). The supported commands are:

- setrot <r> - set frequency correction in milliradians per sample; r is a real number.
- getrot - print current frequency correction
- setchannels - set channels you want to output, format same as with `-c` option, e.g. "1,8,17" will output channels 1, 8 and 17
- getpwr [n] - print relative power of your channels. If an integer n in range $\langle 1, 32 \rangle$ is specified, the transform has n-times higher resolution.

Practical projects

We have created a Tetra receiver using FCL for channelization and provide it in `fcl/examples/tetra.py`. Thanks to the speed of FCL, it is possible to receive significant portion (30 channels) of the Prague Tetra network on a single moderately powerful PC (Intel Core2Quad), and thanks to the telnet interface, it is relatively easy to manage, debug and reconfigure. The receiver also automatically determines and sets the frequency correction.

The developers of [tetrapol-kit] project have ported their multichannel receiver to FCL, so they can reportedly decode 13 channels simultaneously on a Raspberry Pi, while the previous GnuRadio-based receiver allowed 8 channels at most.

2.4.2 Measurements

We have compared the performance of the GnuRadio implementation and our FCL when channelizing Tetra FDM.

GnuRadio 3.7.10 was profiled using `volk_profile` and the time was measured using the `gr_channelizer.py` script (provided in FCL distribution).

The FIR filter for FCL was pregenerated using `./fir.py 1.8e6 18.5e3 1151 rcos > fir.txt` command and the real execution time was measured using time(1) utility. The exact FCL command line was

```
fcl -n 72 -s 50 -f "cat fir.txt" -c 23 -t 4 -i F32 -o /tmpfs/out.ch
```

Three different computers were tested to get an idea about behavior on different systems:

- Embedded system: Raspberry Pi 2 Model B (quadcore ARMv7 with 1 GB DDR2 RAM) running Debian 8, compiled with additional flags `-mthumb--interwork -mfloat-abi=hard -mfpu=neon`
- Laptop: Intel Core i3-2350M (dualcore with HT) with 8 GB DDR3 RAM running Debian 9
- Desktop: AMD FX-8150 (8-core) with 32 GB DDR3 RAM running Debian 8

$54 \cdot 10^6$ samples (30 seconds at 1.8 MS/s) were recorded for the Raspberry Pi and $540 \cdot 10^6$ samples (5 minutes at 1.8 MS/s) for the other two systems. They were stored in tmpfs to avoid interfering with hard-drive access time.

The channelization was executed five times in a row with the first result discarded to eliminate system warmup. The data were then plotted.

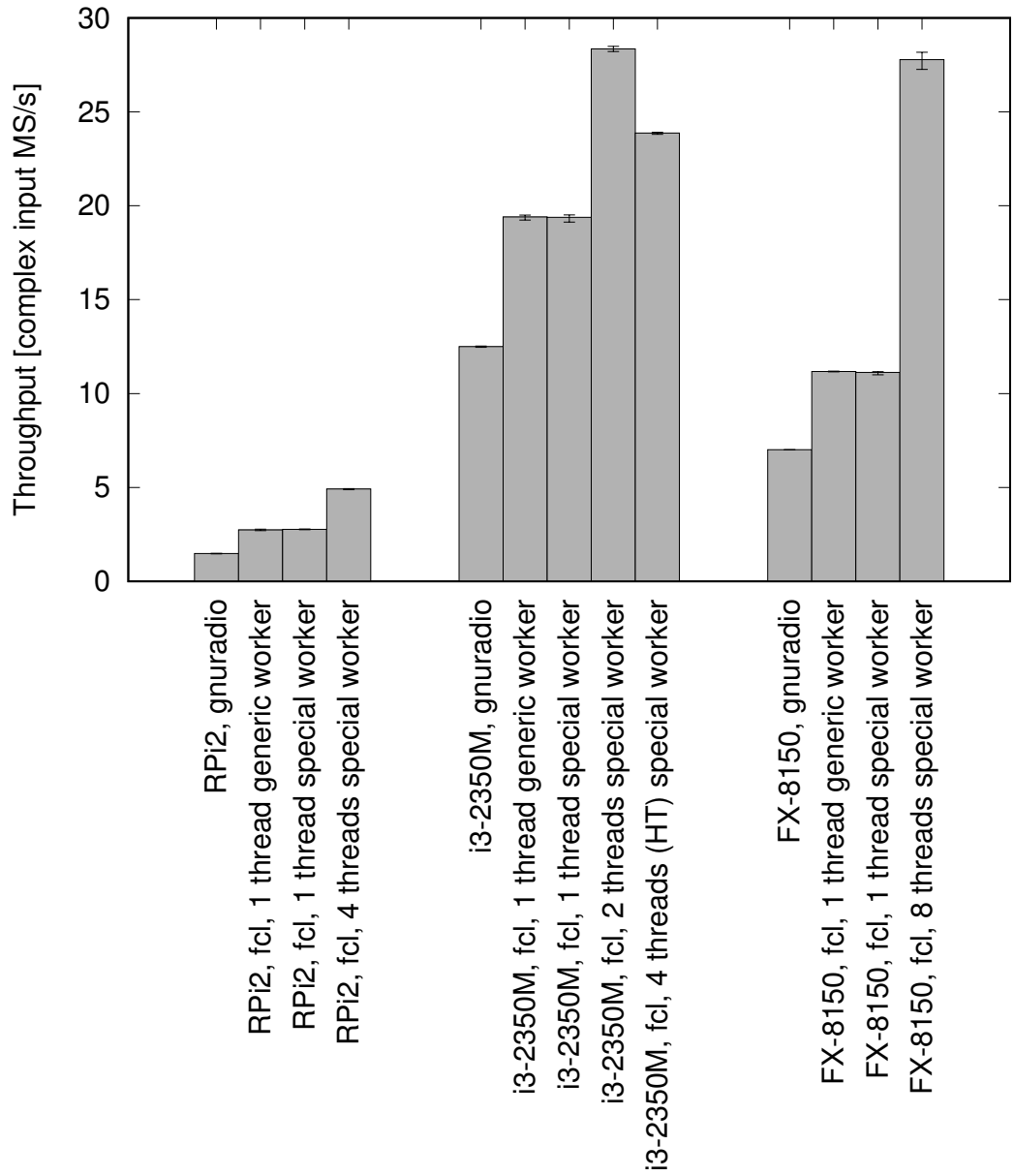

Figure 2.5: Channelizing Tetra (72 channels, 72/50 oversample, 1151 FIR order) with GnuRadio channelizer and with FCL with various multithread and loop unrolling settings, comparing "generic" (loop for any bucket size) and "special" (bucket size specified at compile time) workers. The boxes show average of 4 measurements; the bars show the minimum and the maximum.

We see that FCL is 2 to 4 times faster on these systems.

3. Kukuruku: a client-server SDR application

This chapter describes the design of a SDR application with features proposed in Chapter 1.4.

3.1 Program architecture

Designing the system architecture and user interface posed a challenge. We had experience with common use-cases and tried to capture them, however, several decisions regarding decomposition as well as which features to implement on the client and which on the server had to be made. Finally, we have implemented the following:

- server - a program written in C, reading samples from SDR and implementing commands like "stream channel of a specified width at a given frequency" and "send me spectrum and histogram measurements". The server was intended to be simple and with low performance demands, as one of our use-case is to have a Raspberry Pi or a similar embedded system near the antenna and access it over the network.
- libclient - a Python module to act as a client. It can issue commands and read channels from the server. This allows the user to create custom applications using this infrastructure with simple Python commands.
- kukuruku-gui - a GTK application built upon libclient.

3.1.1 The server

The server uses Python wrapper, `osmosdr-input.py`, around gr-osmosdr to read samples. Thanks to this, it supports wide scale of radios supported by OsmoSDR, but this also means that it depends on complete GnuRadio - a seemingly complicated dependency for something intended to run on embedded systems. Fortunately, as GnuRadio is now in repositories of most popular distributions, it does not require complicated crosscompilation anymore.

The server communicates with this gr-osmosdr wrapper via a named pipe, which is initialized in `init.sh`, and simple messages are passed from the server to the wrapper. The first byte specifies type (`TUNE`, `PPM` or `GAIN`) and then the parameters follow. See `set_param_thr` in `osmosdr-input.py` for the message parser.

Server architecture overview

Upon startup, the server parses command-line options and then spawns one thread that reads from the SDR into a ring buffer, another that reads processed data and writes them to clients and yet another that binds to a socket and accepts connections from clients.

Next, one thread is spawned for each channel we are receiving (this is called *xlater* as it translates the signal from the SDR with a complex heterodyne to zero frequency, low-pass filters it and down-samples it) and one for each TCP client (`client_read_thr`), reading commands and executing them using `parse_client_req`).

Server tuning

The file `constants.h` contains several tunable options:

- `SDRPACKETSIZE` defines how many samples to read and process at once. For SDRs with very high sample rates (>20MS/s) it might be increased to reduce overhead.
- `BUFSIZE` defines how many packets to hold in history buffer. You definitely want to set it so that `BUFSIZE * SDRPACKETSIZE * sizeof(complex64)` fits into RAM.
- `WRITE_FRAMES_SYNC` defines how many packets to write to disk at once when dumping history. As writing a big chunk of data may trigger a long `fsync` and lag the system, it is advised to keep this number small.

3.1.2 Protocol description

Client-server concepts

The server keeps track of channels (in the `worker` list), TCP clients (in the `tcp_cli` list) and mapping which clients wants which channels (the `req_frames` list). The structures are defined in `worker.h` and `socket.h`. The supposed usage of this mechanism is as follows:

- The client creates some channels with the `CREATE_XLATER` message.
- The same client, or maybe some other client, issues the `ENABLE_XLATER` message. The server starts sending samples of the selected channel. Multiple clients can be subscribed to the same channel and one client can have multiple channels subscribed.

This mechanism is intended to allow having "worker clients" that process data and "management clients", maybe with GUI, that set up the channels and then disconnect and leave the architecture running.

Communication protocol

Everything is little-endian. On big-endian machines all messages are automatically converted using functions in `bits.c` on server and little-endian unpacking by Python struct/numpy on client (we have tested this to some extent, but we unfortunately don't have access to a big-endian machine powerful enough to actually run this).

Each message in the stream is preceded by an 8-byte header, of which 4 bytes are message length and 4 bytes message type. Available message types are defined in `client_parser.h`. After the header, payload follows.

The payload are *Google Protocol Buffer* messages except the message that transfers raw data. These messages are defined in `c2s.proto` and `client_parser.h` files. We provide an example communication on Figure 3.1, showing the most important messages.

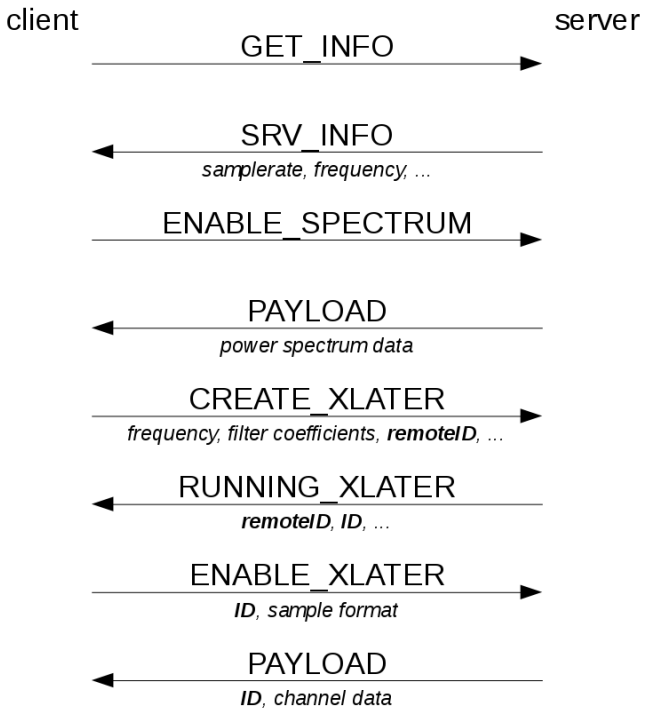

Figure 3.1: An example client-server session

The client connects to the server, requests server information and then requests spectrum data. Once `ENABLE_SPECTRUM` is received, the server streams `PAYLOAD` messages with `payload = SPECTRUM`. They contain array of floats representing power density computed as

$$a_i = 10 \log_{10} \left(\frac{\mathrm{Re}(\mathrm{DFT}(x)_i)^2 + \mathrm{Im}(\mathrm{DFT}(x)_i)^2}{N} \right) \quad [\mathrm{dB}]$$

using this definition of DFT: $\mathrm{DFT}(x)_k = \sum_{n=0}^{N-1} x_n w_n e^{-2\pi i k n/N}$ with Hamming window w. In a similar manner, `ENABLE_HISTO` can be set and arrays of `uint16_t` will be streamed, representing histogram of samples read from SDR.

Of particular interest should be IDs used to reference xlaters. In the `CREATE_XLATER` message, the client sends a remoteID, which is a nonnegative integer chosen arbitrarily. The sever assigns a unique ID and sends this mapping in the `RUNNING_XLATER` message. The xlater is further referenced by this unique ID. In `RUNNING_XLATER` messages sent to other clients remoteID is set to -1. This enables multiple concurrent clients to create xlaters without ambiguity.

Finally, an `ENABLE_XLATER` command is issued and the server begins streaming of the channel data. The interesting parameter in this message is `sample_format`. The native format is a single-precision float. While this provides good dynamic range, it wastes the network bandwidth as 8 bytes are required for each sample (one float for real and one for imaginary component). Therefore, the server can convert the samples into `int16_t` or even `int8_t` (at the cost of introducing

quantization error). The samples are automatically converted back to floats by the libclient library, so the user should not notice anything (except the reduced 8-bit precision when using `int8_t`). Technically, this is accomplished by summing all filter coefficients in absolute values (`calc_max_amplitude` in `xlate_worker.c`) to get the maximum value of a sample a filter can produce and then scaling this number to a target data type.

3.1.3 The libclient library

`libclient.py` is a Python module that provides an interface to the server. The protocol commands are basically mapped to libclient's functions. The available functions have Docstring comments to provide the necessary description.

This short example creates a FM radio listener with some constants hard-coded. A more comprehensible example including the use of callbacks can be found in `cli.py`.

```
from gnuradio.filter import firdes
import libclient
import math

cl = libclient.client()
cl.connect("localhost", 4444)
# automatically subscribe newly created xlaters
cl.set_auto_enable_xlater(True)
# tune the radio to 100 MHz
cl.set_frequency(100000000)
# compute the rotator exponent
rotate = 300000.0/2048000 * 2*math.pi

cl.create_xlater(rotate,
  int(2048000/128000), # decimation
  firdes.low_pass(1, 2048000, 50000, 10000, firdes.WIN_HAMMING), # filter
  "./modes/wfm.py", # binary to pipe the channel to
  -1) # no history
raw_input('...') # wait
```

The library can provide dictionary of xlaters. The dictionary is indexed by ID and contains `XlaterT` objects with the following properties:

- float rotate — rotator in radians per sample
- int decimation — the ratio of sdr_samplerate/channel_samplerate
- bool sql — apply squelch to this channel. When spectrum measurement is received, libclient calls the `sql_callback` function and based on itsreturn value, it either will or will not write channel data to the demodulator stdin.
- bool afc — apply afc to this channel. When channel data are received, libclient computes FFT on them and sums absolute values of upper and lower half of the bins. It then adjusts the xlater frequency to follow the peak of the signal.
- int rid — local reference ID

- threading.Thread thread — feeder thread
- Queue.Queue data — queue of channel data that are written to program stdin
- string sqlsave — in case of closed squelch, this contains the last frame of payload. Then, when the squelch opens, we replay this last frame, so the beginning of the conversation won't get lost.

The dictionary can be obtained by the `get_xlaters` function. If anything requiring concurrent access is to be done with it, the dictionary must be synchronized by calling `acquire_xlaters` and `release_xlaters` to acquire and release the associated lock in libclient.

For more complex applications, several callbacks can be registered within libclient:

- fft_callback — registers a function that is called every time a power spectrum measurement is received. The arguments to the function are list of floats representing the spectrum and timestamp.
- histo_callback — registers a function that is called every time a histogram measurement is received. The argument to the function is a list of integers representing the histogram. Currently its length is 256 and it is not normalized in any way.
- info_callback — registers a function that is called every time an INFO message is received. The argument to the function is an unpacked protobuf INFO message.
- xlater_callback — registers a function that is called every time the xlaters dictionary is changed. If the user of the library is a GUI, it is, for example, expected to update control panels.
- sql_callback — registers a function that is called every time a payload for a channel that has squelch enabled is received. The function is called with arguments of rotator and decimator and it is expected to evaluate whether squelch for this channel should be open and return a boolean. When `False` is returned, the flow of samples from libclient to the demodulator process is paused.

3.1.4 The kukuruku-gui client

A graphical program was implemented using PyGTK for GUI controls and PyGame, a convenient SDL wrapper, for drawing histogram and waterfall.

The program reads configuration files from directory `~/.kukuruku`.

- `~/.kukuruku/gui` – basic parameters, like font size, default server address and preferred sample format.
- `~/.kukuruku/modes` – file containing description of available demoduators. Each demodulator has its own INI section (section is defined by `[SectionName]`) and inside the section are the following parameters:

- `name`
- `program` – the binary that is spawned and raw channel data are piped to its standard input. The format of the data is stream of single-frecision floats, representing I and Q samples.
- `rate` – the required sample rate of the channel. It may actually vary a bit as the server supports only rational decimation - if no integer decimation coefficient is found, the nearest is used.
- `bw` – the required bandwidth of the channel. A low-pass FIR filter with this bandwidth is automatically generated using the GnuRadio Firdes library.
- `filtertype` – type of the filter, currently `hamming` (a Hamming window filter) and `rcos` (a Root Raised Cosine filter) are supported. Most modes are using the Hamming filter, but some digital protocols have Raised Cosine prescribed by specification.
- `transition` – "quality" of the filter. If `filtertype == hamming`, then this is the transition band (from 0 to -60 dB) in Hz. Lower transition band means higher filter quality and higher computational demands. If `filtertype == rcos`, then this is the order of the filter. Higher order means higher filter quality and higher computational demands.
- `resample` – if set to true, then a `-r <float>` parameter is added to the binary, representing the ratio real sample rate / required sample rate, so the demodulator can make corrections according to this (see the description of the `rate` parameter). As our modes use GnuRadio, we supply this parameter to the `gnuradio.filter.fractional_resampler` block. If set to false, then if an exact integer decimation cannot be found, an exception is thrown.

The demodulators provided by default include demodulators for wide and narrow FM and Tetra and Tetrapol networks. We tried to keep them simple, using basic GnuRadio blocks, and we hope they could be used as example demodulators in other applications.

3.2 User guide

3.2.1 Installation

The software is available in the Attachment 2. Both the server and the client can be compiled by simply typing `make` in `kukuruku/server` and `kukuruku/client` directories. The depencencies are FFTW3, VOLK, GnuRadio with Python bindings, protobuf-compiler and protobuf-c-compiler. Additionally, the client requires python-configparser, PyGTK and PyGame, and if you want to decode Tetra and Tetrapol networks, you need to have third-party tools [tetra-listener] and [tetrapol-kit] installed — you need `tetra-rx` and `tetrapol_dump` in `$PATH`.

Unfortunately, shortly before our work was finished, GnuRadio version 3.7.10 was released with a broken `grcc` compiler. We have reported the bug to the developers (the bugfix is awaiting merge) and we provide a patch in `grcc.patch`.

The workaround without applying the patch is to open each `.grc` flowgraph in `gnuradio-companion` IDE and press the Build button. Another bug unfortunately causes GnuRadio to ignore SIGPIPE, so the demodulator won't get closed automatically at the end of the stream.

On the client side, the `.kukuruku` directory containing sample configuration should be copied to the user's home directory and the path to `modes` directory should be edited in `.kukuruku/gui`.

3.2.2 Starting it

On the server, run `init.sh` with the following arguments:

```
server> ./init.sh
Usage: ./init.sh device rate ppm
```

The `device` parameter is the device string recognized by GrOsmoSDR. It is usually in format `radiotype=number` denoting the driver to be used and the device number. For example the first rtl-sdr device on the computer would be `rtl=0`.

The `rate` parameter sets the sample rate in Hz and the `ppm` parameter sets the initial frequency correction in parts-per-million. A tool such as [kalibrate-rtl] or our general abstraction of it, [kalibrate-everything], can be used to measure frequency error of the radio by comparing it to the GSM signal from a base station (it is assumed that a GSM BTS has more accurate clock than especially low-end SDRs).

On the client, run `kukuruku-gui.py` with the argument `server_hostname:port` (the default is 4444 and can be changed in `init.sh`). A window should appear with waterfall and histogram data in it.

3.2.3 Usage

Radio parameters - frequency, gain and frequency correction - can be changed by editing the text boxes in the toolbar and pressing Enter. The frequency field accepts SI prefixes k, M and G like `100M` for $100 \cdot 10^6$ Hz.

You can adjust the gain of the SDR so the histogram looks somewhat balanced and the dynamic range of the SDR is fully used.

Pick a signal on a waterfall and right-click on it to start a demodulator. Once the demodulator is running (a vertical dashed line will be displayed on the waterfall), the frequency can be changed via drag-and-drop and the bandwidth via scrolling. These parameters can also be edited in the bottom panel, which lists all the demodulators. Automatic frequency correction and signal squelch can be activated there too.

If you right-click on an existing demodulator in the waterfall, instead of creating new xlater on the server, this existing one is subscribed.

If you hold CTRL while right-clicking on the waterfall when creating a new xlater, it will start the channel from that point in the history buffer.

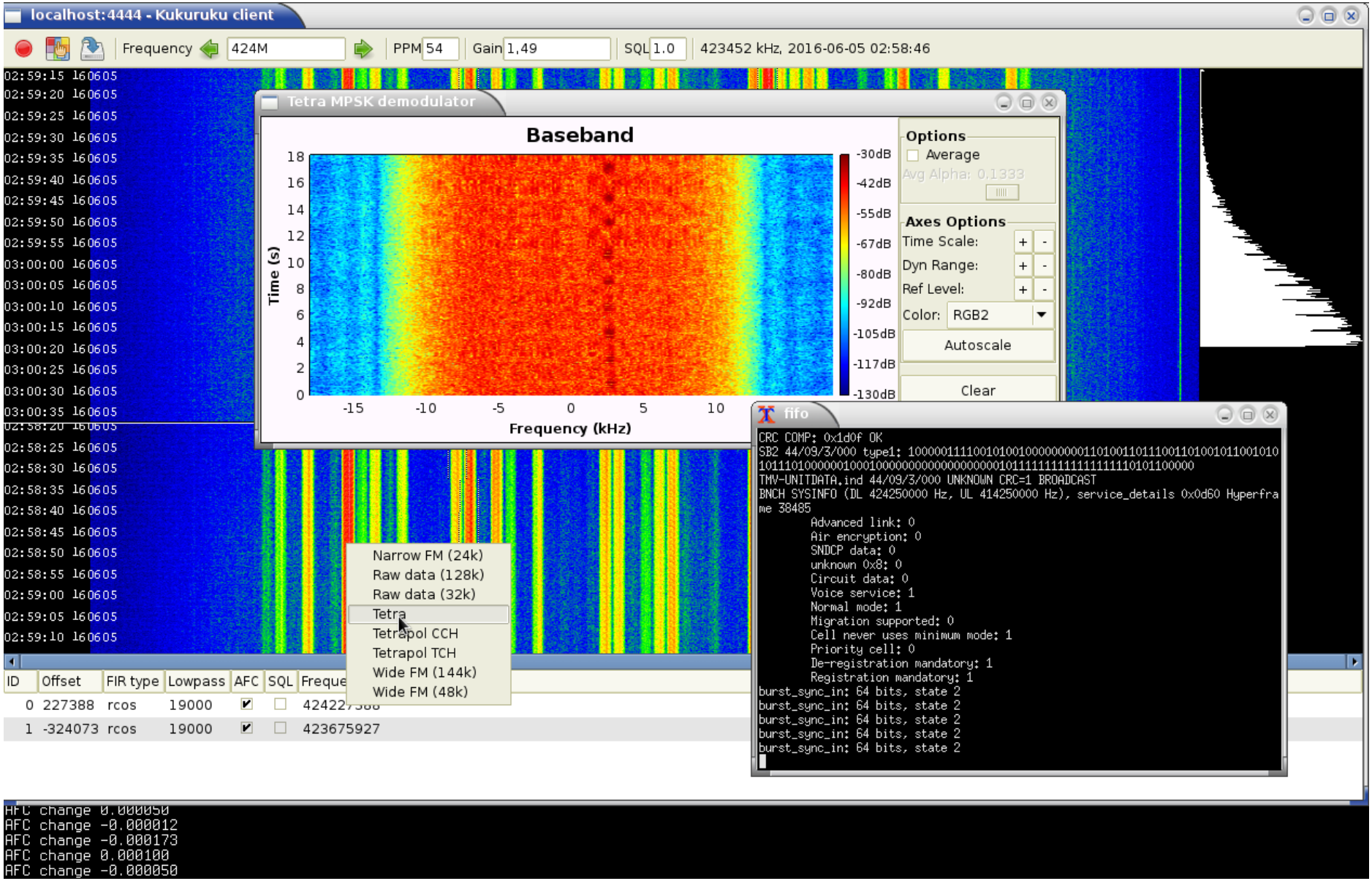

Figure 3.2: A kukuruku-gui screenshot

4. The scanner

With this tool, the user can specify frequency ranges he is interested in, and the SDR is then periodically automatically pseudorandomly retuned. If some signal is present, it is dumped into a file. It is possible to specify how long the signal should be recorded after it has disappeared (e.g. there could be an expected pause of speech on analog trunked networks) and it is possible to schedule recording of a given channel using a crontab-like format ("record frequency associated with sirens every Wednesday noon"). The scanner does not visit the specified frequencies sequentially or randomly, but based on the result of `SHA(timestamp + nonce)`. With this, one can have multiple scanners with time synchronized over NTP scanning the same bands from different locations. Unfortunately, as consumer-grade SDR hardware supporting precise timing synchronization is not readily available, one cannot determine the location of the signal source using multilateration.

4.1 Program architecture

The program is available in Attachment 3 and consists of `KukurukuScanner` Python module, `xlater.c` Swig module, which implements the algorithm described in Section 2.1, and `sorter.py` script, which allows simple categorization of recorded signals. The Swig module must be compiled by running `make` with Swig3 being the dependency.

The main script `scanner.py` firsts uses `util.py` to read the configuration and then spawns gr-osmosdr thread which reads samples from SDR and `KukurukuScanner` thread which runs the main loop.

4.2 Configuration

The configuration is in the form of several `*.conf` files in the `~/kukuruku/scanner` directory.

There are two special files named `main.conf` and `blacklist.conf`. The `main.conf` file contains general configuration and is richly commented. The `blacklist.conf` file contains list of frequencies which should be either ignored or at least activity on them should not be regarded as a newly found signal. Its format follows:

```
<i or a> <frequency> <bandwidth> [optional comment]
```

Each line denotes frequency interval

$$[\text{frequency} - \text{bandwidth}/2, \text{frequency} + \text{bandwidth}/2]\,.$$

If the first column is `i`, signals that have center frequency in this interval are completely ignored (e.g. non-interesting radio stations); if it is `a`, signals are directly archived (recorded to the `archive/` directory), so `sorter.py` does not prompt the user about them.

The `main.conf` file contains the following important options (and some more, which are documented in comments inside the configuration file):

- `samplerate` — The SDR sample rate.
- `nonce` — A salt that is hashed together with current timestamp to determine the next frequency when random scanning is in progress.
- `stick` — The number of seconds to record the active channel for.
- `silencegap` — The number of seconds there must be no signal on the channel to be treated as inactive (e.g. if it is an analog speech, gaps are expected).
- `stickactivity` — If set to `yes`, the scanner will continue to record the channel even if `stick` seconds already elapsed if the channel is still active (i.e., the signal level was above squelch threshold in the last `silencegap` seconds).
- `messgain` — Specifies which of the SDR gains should be used for autogain. The scanner computes histogram every time it retunes and adjusts the SDR gain based on it. Note that these adjustments happen on per-frequency basis, which is important because the antenna and the input stage may not have the same gain on all frequencies.

All the other `*.conf` files are supposed to contain description of channels or frequency ranges we want to scan. They contain one `[General]` section and optionally sections specifying channels.

The `[General]` section contains either `freqstart` and `freqstop` options, then the file specifies frequency range, or only the `freq` option, and then the file specifies a single frequency which is used as a center frequency for the SDR. All the frequencies are specified in Hz and SI prefixes `k`, `M` and `G` are supported.

The `[General]` section may contain the `stick`, `stickactivity` and `silencegap` options. The values of these options from `main.conf` are then overriden for this particular frequency or range.

If the file specifies a single frequency, the following options can be specified:

- `cron`, containing crontab-like string (in format `minute hour day_of_month month day_of_week`) specifying when the frequency should be tuned to. Single number expressions, `*` (meaning *always*) and `*/N` (meaning *when divisible by N*) are supported.
- `randscan`. If set to `no`, the frequency is tuned to based only on the contents of the `cron` option – it is not scanned pseudorandomly.

Then, again for the single-frequency file, a list of channels follows. Each channel has its own section (started by `[ChannelName]`) and two or three parameters:

- `freq`, its frequency. abs(channel frequency − frequency specified in config file) must be less than half the SDR samplerate (so the channel is "visible" to the SDR).
- `bw`, the channel width.

- `pipe` (optional). If specified, instead of saving raw I/Q samples to a file, the command is executed. If the command contains the string `_FILENAME_`, it is replaced by frequency and timestamp.

Let us show two examples of these configuration files.

```
[General]
freqstart=445000
freqstop=449000
```

This file specifies a single frequency range 445 to 449 MHz. The range is then broken into overlapping chunks and the radio is pseudorandomly tuned to them. For example if the SDR sample rate is 2 MHz and `overlap` (in `main.conf`) is set to 0.2, the chunks will be

$$\{(444.8, 446.8), (446.0, 448.0), (447.2, 449.2)\} \text{ MHz}.$$

The next example demonstrates scheduled recording.

```
[General]
freq=100M
cron=0 12 * * *
stick=300
randscan=no
stickactivity=no

[Channel1]
freq=99700k
bw=144k
pipe=my_wfm_demodulator.py _FILENAME_
```

First we instruct the scanner to tune to 100 MHz, but not randomly (`randscan` set to `no`), but only every day at 12 o'clock. We then record for 300 seconds and we don't continue recording even if the signal is still present after 300 seconds (`stickactivity=no`). Finally, we don't save raw I/Q samples into file, but pipe them to stdin of our program instead.

4.3 Usage

The `scanner.py` script is simply executed with optional arguments `-d`, `-p` and `-o` specifying OsmoSDR device string (e.g. `rtl=0` for the first rtl-sdr device on the system), frequency correction in parts per million and the output directory. The program begins scanning and saves the recordings to the output directory.

4.4 Sorting the signals

The `sorter.py` script takes recorded files from the current working directory (or files given as command line arguments) and displays menu, where the user can demodulate the signal using modes from Kukuruku client (described in Section 3.1.4) and decide whether to keep or delete the recording and whether to add the frequency to a blacklist.

Conclusion

We have described important algorithms for selecting one or more channels from a wideband received signal and we have then used them to build several applications that we considered were missing, be it a high-performance channelizer or a univesal receiver. We have already seen FCL in real deployments. And even if the Kukuruku client-server SDR software won't get widespread adoption, we consider it important to show features, both of the user interface and the flexible network architecture, that could be implemented into other similar software projects.

Future work

In the Section 2, we have concluded that our channel selector and channelizer are "fast enough" for our current applications (which are namely receiving Tetra and Tetrapol networks). Should the need for better performance arise, one can implement further optimizations, for example selecting between evaluating FIR by definition and by FFT overlap-add method. The same case is with the channelizer — one can probably devise more clever methods of partitioning work between threads that don't require moving too much data in memory.

We have already used Kukuruku in several practical use-cases. For future developments, two features should be considered:

- Server-side demodulators. The bandwidth could be saved even more if even the narrowband channel is not transmitted. This means having the demodulator on the server and transferring only demodulated data, e.g. bits in digital modes or audio (further compressed with some lossy codec).
- Server-side AFC and squelch, which may again save some bandwidth.
- Dynamic waterfall zoom. Currently the server can send arbitrary FFT size, but this cannot be smoothly zoomed at the runtime. Implementing this to be bandwidth-efficient might be tricky as currently we are transmitting floats, i.e., 4 bytes for each pixel. Possibilities to use some image compressing technique (DCT, wavelet) should be investigated.
- Autogain by analyzing the histogram, in a similar way the scanner has.
- Extending the protocol by some kind of "privileges", which specify users that are allowed to tune the SDR, or only receive channels etc.

The scanner is pretty simple as of now and may receive the biggest attention. Both the backend (more intelligent peak detection — there are lots of heuristic algorithms to investigate), the frontend (one can imagine a GoogleMap-like interface showing gigantic waterfall collected over several days with recorded signals overlaid) and even the hardware (if we somehow manage to smuggle time-synchronization signal into the baseband, we could build an experimental multilaterating system) may be improved.

Our three programs are mostly independent. Depending on future developments, we may unify some duplicate functionality (like SDR retuning and the channel selector), or even create an abstraction layer that would help sharing one or multiple SDRs between these programs.

Bibliography

GnuRadio. http://gnuradio.org/. Accessed: 2016-05-22.

Gqrx. http://gqrx.dk/. Accessed: 2016-05-22.

gr::filter::pfb_channelizer_ccf. gr::filter::pfb_channelizer_ccf. URL http://gnuradio.org/doc/doxygen-v3.7.10/classgr_1_1filter_1_1pfb__channelizer__ccf.html.

Fredric J. Harris. *Multirate Signal Processing for Communication Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0131465112.

kalibrate-everything. https://brmlab.cz/user/jenda/kalibrate-everything. Accessed: 2016-05-22.

kalibrate-rtl. https://github.com/steve-m/kalibrate-rtl. Accessed: 2016-05-22.

Klusáček, David. NPFL109 Digital Sound Processing. University Lecture, 2015.

Nutaq. Implementation of Gardner symbol timing recovery in System Generator. http://www.nutaq.com/blog/implementation-gardner-symbol-timing-recovery-system-generator. Accessed: 2016-07-23.

tetra-listener. https://brmlab.cz/project/sdr/tetra. Accessed: 2016-07-22.

tetrapol-kit. https://brmlab.cz/project/sdr/tetrapol. Accessed: 2016-07-22.

List of Figures

List of Abbreviations

ADC Analog to digital converter

AFSK Audio frequency shift keying, a method of transfering data over radios originally intended for voice

AM Amplitude modulation

APRS Automatic Packet Reporting System, a data network using AFSK modulation

DAC Digital to analog converter

DCT Discrete cosine transform

DFT Discrete Fourier transform

FCL Fastest Channelizer in Litoměřice

FDM Frequency division multiplex

FFT Fast Fourier transform, an algorithm that performs DFT in $O(N \log N)$ time

FFTW Fastest Fourier Transform in the West, a software library implementing the FFT algorithm

FIR Finite impulse response

FM Frequency modulation

FPGA Field-programmable gate array, a chip whose internal logical connections can be changed by software

GUI Graphical user interface

HT (Intel) HyperThreading

LAN Local area network

PFB Polyphase filterbank

SDL Simple DirectMedia Layer, a multimedia library

SDR Software defined radio

VHF Very high frequency, radio band between 30 and 300 MHz

Attachments

1. `fcl/`, the channelizer described in Chapter 2. This is the snapshot of repository `https://jenda.hrach.eu/gitweb/?p=fcl`.
2. `kukuruku/`, Kukuruku interactive SDR client and server described in Chapter 3. This is the snapshot of repository `https://jenda.hrach.eu/gitweb/?p=kukuruku`.
3. `kukuruku/scanner`, the scanner described in Chapter 4. This is from the `https://jenda.hrach.eu/gitweb/?p=kukuruku` repository too.
4. `grcc.patch`, a patch fixing broken `grcc` in GnuRadio 3.7.10. The bugreport is available at `http://gnuradio.org/redmine/issues/927`.

For production purposes we recommend obtaining the latest versions from the aforementioned repositories as we plan to continue to maintain our software.