OpenWebRX: SDR Web Application for the Masses

András Retzler, HA7ILM

Department of Broadband Infocommunications and Electromagnetic Theory,
Budapest University of Technology and Economics, Hungary

randras@sdr.hu

***Abstract*—Software Defined Radio technology is getting more and more popular among amateur radio operators and hobbyists, as different universal SDR hardware devices have become available recently. OpenWebRX is a software made for those who want to set up remote SDR receivers accessible from the web. It has been developed with open-source codebase, multi-user access and easy setup in mind, to be an alternative to other similar projects (WebSDR, ShinySDR, WebRadio, etc.) It supports cheap RTL2832U based tuners. Basically OpenWebRX is an on-line communications receiver for analog transmissions (AM/FM/SSB/CW), with a web UI on which real-time waterfall display is available. Users can select a channel within the bandwidth of the sampled signal acquired from the SDR hardware. The selected channel is demodulated on the server and the resulting audio is streamed to the browser of the user, where it is played back. Users can set receiver parameters (channel frequency, modulation mode, filter bandwidth) independently. The digital signal processing functions have been placed in a separate library, *libcsdr*, which can also be considered useful as a standalone package. It performs the digital downconversion, filtering and demodulation tasks on the I/Q data.**

Keywords—SDR, RTL-SDR, open-source, HTML5

I. INTRODUCTION

The speed of digital computing is increasing continuously. In addition, in the last years reasonably fast A/D and D/A converters have become available. It has become feasible to implement most of the signal processing digitally in radio systems.

Nowadays Software Defined Radio is widely used in commercial systems from satellite communication to mobile telephony, but it mostly does its task hidden in the background, replacing the conventional analog communications equipment. These embedded SDRs are tailored for a specific need (for example running a 4G base station), producing better results than their analog competitors and solving advanced problems at a lower price.

However, there is also a need for universal SDR hardware for prototyping, measurements and for use in real-world projects. While devices like the USRP, HackRF, BladeRF have come to the market to support professional and academic users, a lot of amateurs have also started to experiment with SDR technology, building cheap receivers for specific bands. When the RTL-SDR project [1] has been released in 2012, even more people got in connection with SDR directly. A mass-produced DVB-T receiver USB dongle with an RTL2832U chip is a great entry-level device that can be acquired for $15, and with the help of the RTL-SDR project, it is possible to use it as a general purpose SDR. It tunes between 24 MHz and 1700 MHz, can be used as a receiver for a variety of wireless equipment, from two-way radios to wireless temperature sensors, and it works on multiple amateur radio bands as well. There is a vast number of free software available for decoding different signals with it. Although its professional competitors provide better performance, it is still an ideal choice for the experimenters.

RTL-SDR is also a great educational tool that can help to get more young people into amateur radio. For those, who already have the required background in IT, playing with the cheap DVB-T stick is a great chance to dive into the radio waves. I even had a friend under the age of 15 who managed to receive the signals of an amateur radio satellite with RTL-SDR on his own.

Two years ago I released an open-source project, SDRLab [2], which implements an RTL-SDR interface for LabVIEW along with a simple WFM demodulator example, and since then, I received e-mails from a few university students from different countries, asking about it. It turned out that at some universities, RTL-SDR is used as an educational tool to teach digital signal processing.

So why not bring SDR closer to even more people? The motivation behind OpenWebRX was to create a remote SDR receiver that is available from the Internet, so that amateur radio operators can connect to the server with a web browser, click on an interesting signal on the waterfall display, and listen to the demodulated audio stream, without the need of purchasing any dedicated hardware to experiment with SDR (as shown on Figure 1).

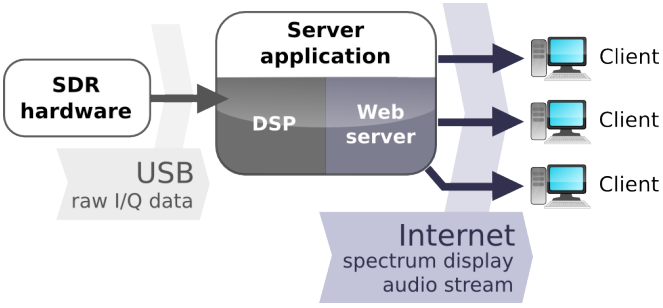

Fig. 1. A high level block diagram of OpenWebRX

II. Alternatives and Design Considerations

The idea is not new: there are several software available for this purpose. The best known is WebSDR, by Pieter-Tjerk de Boer, PA3FWM. WebSDR has been developed since 2007, and now has numerous great features, like the auto-notch filter and the HTML5 mode which also works on mobile devices. The drawback is the availability of the software: as stated in the WebSDR FAQ, you can only get it directly from the author, so it is not distributed to anyone by means of free download, and the source code is not available at all, so even if you have the binaries, you cannot make your own modifications. Another good alternative is ShinySDR by Kevin Reid, AG6YO, which is open-source under the GPL license, but currently more suitable for use by only a single person at the same time (Kevin noted that multi-user support will be available in the future). One more notable project is WebRadio by Mike Stirling, which is not actively developed at the moment.

When I started working on OpenWebRX, only WebSDR existed as an alternative. I wanted to create a software package that implements an SDR receiver that satisfies the following requirements:

- it has a web UI (shown on Figure 2),
- it works with RTL-SDR dongles,
- multiple people can use it at the same time, and they can tune to different frequencies independent to each other, within the receiver bandwidth,
- the source code is available under an open-source license, as I consider reviewing, modifying and improving ham radio gear and software as a good practice.

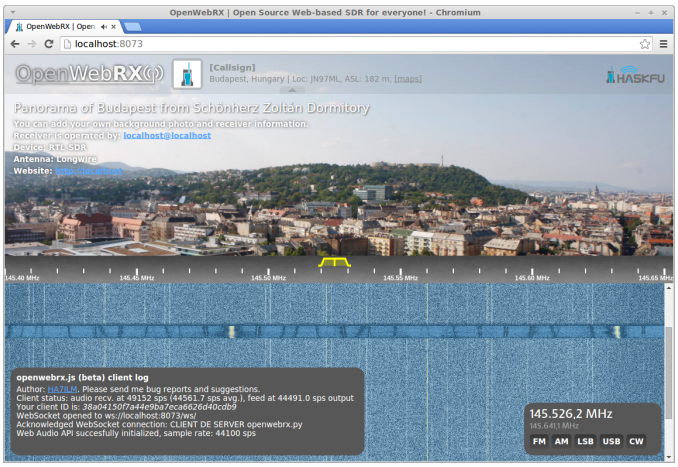

Fig. 2. A screenshot of the OpenWebRX web UI

III. Setting up an OpenWebRX server

To set up an OpenWebRX server at home, you will need a computer running Linux, and an RTL-SDR dongle. OpenWebRX has been tested on Debian/Ubuntu based systems, but it should work on other Linux distributions as well. If the dependencies have already been installed (*rtl-sdr, libfftw3-dev, git, python 2.7, gcc, bash*), then the first step to do is to get the git repositories:

```
git clone \
https://github.com/simonyiszk/openwebrx.git

git clone \
https://github.com/simonyiszk/csdr.git
```

Next, *libcsdr* should be compiled:

```
cd csdr
make \
sudo make install
```

Now OpenWebRX can be started. If it is ran with the default settings, it will set the RTL-SDR to the 2-meter band, and begin acquiring samples.

```
cd ../openwebrx
./openwebrx.py
```

The receiver can be opened in the web browser by typing the following URL:

http://localhost:8073/

As OpenWebRX uses some recent HTML5 features that are not present in older browser versions, it requires the latest Google Chrome or Mozilla Firefox. It should also work on Chrome for Android, although it is not optimized for mobile devices yet.

By editing the *config_webrx.py* file, receiver settings can be configured (center frequency, sampling rate, gain) and also the details shown on the web UI (callsign, location, antenna, etc.) can be customized.

IV. Using OpenWebRX

The user interface of OpenWebRX is similar to other SDR software: it has the appropriate buttons for changing modulation, receiver frequency can be set by clicking on a signal shown on the waterfall display. An additional feature is the availability of the whole history of the waterfall display: one can go back in time with the help of the scrollbar on the right edge of the window. The filter bandwidth can be changed by dragging the ends of the yellow filter shape shown above the frequency scale (see Figure 3). Additionally, to achieve the same effect as turning the passband shift (PBS) or the beat frequency oscillator (BFO) knob on a real radio, the filter shape or the VFO tick should be dragged with the mouse, while holding the shift key down.

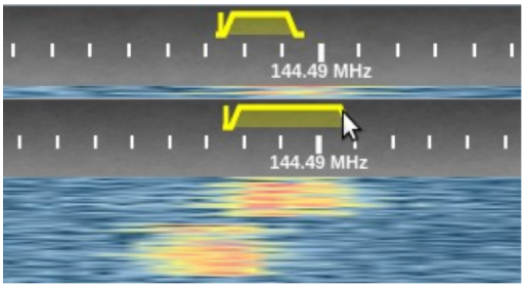

Fig. 3. Changing the filter bandwidth

V. Performance notes

One of the greatest challenges in today's SDR receivers is the processing of the high amount of acquired data in real time. Even RTL-SDR is capable of bringing 2.4 Msps of 8-bit I/Q data into the computer, which means 4.8 megabytes of raw data to process every second. It is not a problem if you want to downconvert and demodulate a signal for a single user, but otherwise the CPU time required to process a block of data gets multiplied by the number of users. If too many clients are connected to an OpenWebRX server, and the CPU cannot handle the processing tasks for all of them, they will get lagging audio. Therefore it is important to optimize the settings to find the compromise between the high receiver bandwidth and the high number of users. There are several settings you can tune in *config_webrx.py*:

- *samp_rate*: The sampling rate directly affects the CPU usage per client. The lower it is, the more clients can be served simultaneously. For *rtl-sdr*, some valid values are 250000, 1024000, 2048000, 2400000.

- *fft_size*: The waterfall display is calculated only once for all clients, but decreasing its resolution can still improve performance (and results in lower network usage as well).
- *max_clients*: It is important to set this value to the safe maximum that will not result in lags.

You can stress-test OpenWebRX by opening your URL in a lot of tabs simultaneously, preferably on another computer. Note that doing this requires sufficient CPU performance at the client as well. On the server, you can check CPU usage with the *top* command.

You do not necessarily need a desktop PC to act as a server: you can also use one of the popular single-board computers (SBC) based on ARM processors. These cheap, small boards have low power consumption. It was reported by several users that the Raspberry Pi 2 is capable of running OpenWebRX, while the original, single-core Raspberry Pi is not sufficient. My tests on the Odroid-C1 board showed that it can serve at least 10 clients when the sampling rate is 240 ksps, so it would be all right for sound-card based SDRs as well. The audio started to lag when the number of clients exceeded 15.

On the other hand, running OpenWebRX on a quad-core Intel i7 CPU can serve 10 clients without problem at the RTL-SDR maximum sampling rate, 2.4 Msps, which is equivalent to processing 48 megabytes of data every second. This is not a bad result, regarding that we are talking about a general purpose processor. The CPU-critical process is the digital downconversion (DDC), which involves frequency translation and downsampling. Other DSP operations (like the actual demodulation and the AGC) work on a relatively low sample rate signal, so they do not take much CPU time.

VI. System Overview

In the following part, the components of OpenWebRX are explained. The two main parts are the server application and the front-end running on the client computer (shown on Figure 4).

The server has been implemented in *python*, a very flexible scripting language, which has definitely helped to cut down the time required for development. Along with running the web service, the server spawns several processes which communicate via OS pipes and TCP sockets with each other and the server core.

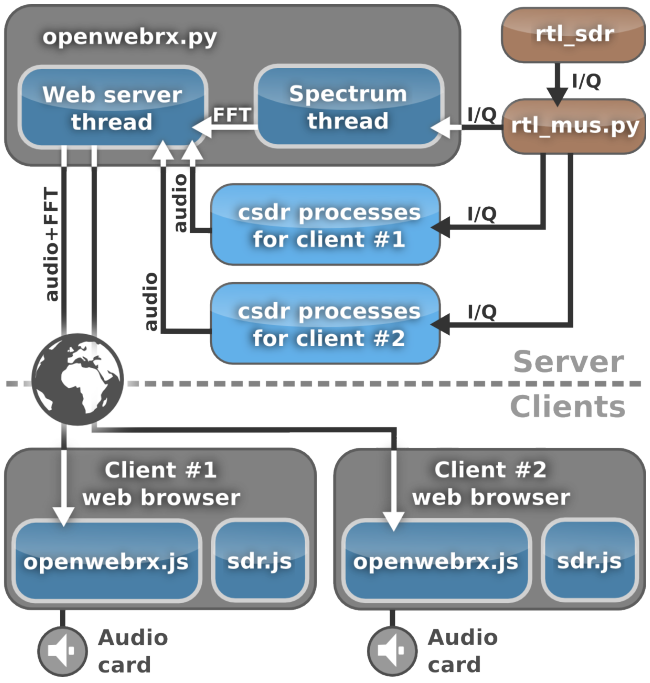

Fig. 4. A more detailed block diagram of the system

One of these processes is the well-known *rtl_sdr* command-line tool, which acquires the samples from the RTL-SDR device. In fact, it can be substituted with any other command that can emit I/Q samples to the standard output, so adding support for other SDR hardware should be easy, one just needs to change the command specified in *config_webrx.py*.

The component used for distributing I/Q data between processes is *rtl_mus.py,* which stands for RTL Multi-User Server. I released this previously as a separate open-source tool. It acts as a TCP server to stream raw I/Q data to multiple clients. (The version packaged with OpenWebRX has the

raw I/Q data port restricted to local connections only, for use by other server components.)

Every time a new client opens the webpage of the receiver, the browser initiates a WebSocket connection, which can be used for efficient two-way communication between the web server and the client (unlike traditional HTTP requests).

On the server side my own WebSocket implementation is used, which resides in *rxws.py*. When the WebSocket connection is established, the server spawns a set of new *csdr* processes as a signal processing chain, which takes its input from *rtl_mus.py,* and outputs the demodulated audio. The audio is then sent through the WebSocket by the server core, along with the spectrum data for the waterfall diagram. The spectrum data is emitted by the spectrum thread, which is common for all clients and thus has only one instance.

The front-end application running in the browser has been implemented in HTML5 and JavaScript: *openwebrx.js* manages the UI and the WebSocket, draws the waterfall diagram, and outputs the audio to the sound card using the Web Audio API. But before the audio can be output, some additional signal processing steps should be taken at the client side, which are performed by *sdr.js*.

VII. LIBCSDR

The DSP functions behind OpenWebRX have been implemented as a standalone library called *libcsdr*. There are already existing software packages for this: GNU Radio is the most notable, which is used by many SDR software for Linux, e.g. gqrx, ShinySDR. One of the reasons why I chose to implement my own lightweight solution is that compiling the latest version of GNU Radio from source takes a lot of time, and it is sometimes difficult to do. *libcsdr* contains only the required functions for AM/FM/SSB/CW demodulation, and should compile in a couple of seconds, even on SBCs.

I optimized the code for the auto-vectorization feature of the GNU C Compiler, so some DSP functions can make use of the SIMD instructions in the CPU, although much more speedup could be achieved by hand written assembly code.

VIII. CSDR, A COMMAND-LINE TOOL FOR ANALOG DEMODULATION

Another feature of the DSP library is the *csdr* command-line tool, which can be used to build simple signal processing flow graphs with one-line Linux commands. This is the wrapper around *libcsdr* that OpenWebRX uses.

With *csdr*, prototyping a headless ham receiver on an ARM-based SBC might be quite straightforward. Example commands for demodulating NFM/AM/SSB can be found on the *csdr* GitHub page [3]. For the sake of simplicity I rather present a one-liner for demodulating a WFM broadcast at 100.2 MHz:

```
rtl_sdr -s 240000 -f 100200000 -g 20 - | \
csdr convert_u8_f | \
csdr fmdemod_quadri_cf | \
csdr fractional_decimator_ff 5 | \
csdr deemphasis_wfm_ff 48000 50e-6 | \
csdr convert_f_i16 | \
mplayer -cache 1024 -quiet -rawaudio \
samplesize=2:channels=1:rate=48000 \
-demuxer rawaudio -
```

In this example, to demonstrate the headless functionality, the raw I/Q data is taken from the output of the *rtl_sdr* tool, and at the end the demodulated audio is fed into *mplayer,* to play it on the sound card without the need of OpenWebRX.

We perform the following processing steps in separate processes:

- **`rtl_sdr -s 240000 -f 100200000 -g 20 -`**

  RTL-SDR outputs I/Q samples with a sampling rate of 240 ksps.

- **`csdr convert_u8_f`**

  First we convert the 8-bit unsigned samples to floating point.

- **`csdr fmdemod_quadri_cf`**

  We run a quadricorrelator FM demodulator, which turns our complex input signal into a real output signal.

- **`csdr fractional_decimator_ff 5`**

  We decimate the signal by a factor of 5, while also running a filter on it to suppress the possibly overlapping high frequency components. We get a 48000 sps signal at the output, which matches the sampling rate of our sound card.

- **`csdr deemphasis_wfm_ff 48000 50e-6`**

  We apply a de-emphasis filter with a time constant of 50 μs.

- **`csdr convert_f_i16`**

  We convert the floating point real samples to 16-bit signed integers to match the format required by the sound card.

- **`mplayer -cache 1024 -quiet -rawaudio (...)`**

  We output the samples to the sound card.

As the DSP functional blocks run in separate processes, the flow graph can take advantage of multiple CPU cores, if available. Scheduling is handled by the operating system. I was skeptic for the first time I have tried this, as I know the importance of scheduling in a dataflow system. While experimenting with OpenWebRX and *csdr* together, it turned out that this solution does quite good job even if about 50 processes are started when multiple users are present. In addition to multi-core processing, this implementation makes the flow graphs of OpenWebRX easily modifiable, as they are defined as a few lines in *plugins/csdr/plugin.py*, and also keeps *libcsdr* code simple.

IX. Experiments with client-side DSP

The first release of OpenWebRX server required about 2 Mbit/s uplink network bandwidth per client, because I sent the 16-bit 44100 Hz raw sample stream through the WebSocket. In order to make OpenWebRX suitable for hams who want to share their receiver from their home Internet connection with a low upload speed, I added two processing steps before sending the data over the network connection:

- I decreased the sampling rate to 11025 Hz, which is still enough for NFM and SSB transmissions.
- I applied IMA ADPCM compression to the audio.

As a result, the required uplink bandwidth now is about 200 kbit/s with the default settings, of which about 70 kbit/s is the audio, and 130 kbit/s is the spectrum data.

In order to restore the original signal at the client side, I had to implement the reverse operations in JavaScript.

JavaScript has developed much in the last few years. It was traditionally an interpreted language, which resulted in with very poor execution speed, but with the introduction of JIT compilers, namely TraceMonkey in Mozilla Firefox and the V8 Engine in Google Chrome, JavaScript is now feasible to build complex applications on.

Some of the new, and still not widely known achievements on the scene of JavaScript are the Emscripten compiler and *asm.js*. The Mozilla Foundation has designed *asm.js* to be a subset of JavaScript that JIT compilers can easily optimize for, so performance is typically 50-67% compared to the speed of the native version of the same code [4]. Emscripten is an LLVM-based, source-to-source compiler to translate C/C++ applications into the *asm.js* subset of JavaScript. For example, companies use it to port games to the web, by compiling the original C++ code of the game (that was linked against OpenGL) into JavaScript code that uses WebGL, so the game can be played directly in the browser.

Instead of porting some algorithms from *libcsdr* from C to JavaScript manually, I decided to compile the entire *libcsdr* package to JavaScript with Emscripten. The resulting JavaScript library has been called *sdr.js*, and contains all the functions that *libcsdr* has. However, there are some drawbacks. First, a lot of additional wrapper code had to be written to actually use the compiled functions. At this point only functions that are essential to OpenWebRX (like the IMA ADPCM codec and the resampler) have wrappers yet.

It is important to note that here I do only some post-processing on a signal with relatively low sample rate (<50 ksps), so the 50% performance difference between JavaScript and the native code is not a problem. However, if someone would want to write a standalone SDR application running in the browser, the speed of the DDC operation and the FFT calculation would be limited compared to a native implementation, and also SDR hardware access is complicated from the browser (although Google's Radio Receiver application for Google Chrome, which has entirely been implemented in JavaScript, works in a similar way).

Back to the compression, it is also worth to mention that the spectrum data is compressed, too. As it is a one-dimensional vector of dB values, it was not feasible to use an image compression algorithm like JPEG. I've had the idea to test whether ADPCM works on it, as the spectrum data is just like any other real-valued signal, and not that different from an array of audio samples. Surprisingly, I had good results, so I kept using it.

X. Real-world uses of OpenWebRX

OpenWebRX is currently used by only a few stations.

Richard van der Riet, PA3GWH and Bart Weerstand, PA3HEA have been running their servers since February. Richard's server is monitoring the 30 meter band, while Bart's is currently working on 2 meters.

With the help of Levente Dudás, HA7WEN I had the chance to do some testing on one of the machines of HA5MRC Radio Club in Budapest, with a receiver connected to a Yagi antenna array automatically pointed to the JAS-2 amateur satellite during its pass.

Antal Vincze, HG4FC was the first to test the OpenWebRX server in production environment at HA5KAW club station in Nadap, Hungary.

John Seamons, ZL/KF6VO has adapted OpenWebRX for use with his direct sampling HF receiver.

Alex Trushkin, 4Z5LV has made a version of OpenWebRX that is compatible with AFEDRI SDR.

A few days before submitting the paper I have got the news that the first OpenWebRX server on HAMNET has become available, by Hans Reiser, DL9RDZ, and there is one more to be set up soon.

XI. Conclusion and future plans

Today's technology allows us to implement complex applications like an SDR receiver as a web service. A multi-user SDR receiver requires much more CPU performance than a single-user application, still we can find a compromise to let the application run on single board ARM computers.

As young people of today have born into a world with computers and mobile phones, I hope that web receivers will make some of them interested in the great hobby of amateur radio.

If you are further interested in the OpenWebRX project, you can find more information at the following website:

http://openwebrx.org/

On this page, my Bachelor's thesis that I have written on this topic is also available for download. It contains more details about the implementation of both the web service and the signal processing. I hope it will help those who want to write the code for their own ham radio SDR receiver.

The development has not stopped after submitting my thesis. The website for listing the receivers is expected to be ready soon. Users have requested many improvements, for example a squelch, and support for tablets, so there is a lot of things to do. The DDC could be much more optimized in order to be able to serve more clients from the same host, and using GPU hardware acceleration is also under consideration.

XII. Acknowledgements

I would like to thank to everybody who helped me with this project, and especially my friends János Selmeczi, PhD (HA5FT) and Péter Horváth, PhD (HA5CQA) for their continuous help and support.

References

[1] OSMOCOM (2015. 07. 15.), *rtl-sdr.* Available: http://sdr.osmocom.org/trac/wiki/rtl-sdr

[2] András Retzler (2015. 07. 15.), *SDRLab: an RTL-SDR interface to LabVIEW for educational purposes.* Available: http://ha5kfu.sch.bme.hu/sdrlab

[3] András Retzler (2015. 08. 16.), *csdr.* Available: https://github.com/simonyiszk/csdr

[4] Alon Zakai (2015. 08. 16.), *Emscripten & Asm.js: C++'s Role in the Modern Web* (slide 26). Available: https://kripken.github.io/mloc_emscripten_talk/cppcon.html#/26