An Over-the-Air Reconfiguration API for Experimental Cognitive Radio Setups

Moritz Fischer, Martin Braun, Jens P. Elsner and Friedrich K. Jondral
Communications Engineering Lab, Karlsruhe Institute of Technology (KIT), Germany
moritz.fischer@student.kit.edu, {martin.braun, jens.elsner, friedrich.jondral}@kit.edu

***Abstract*—We present a software solution for over-the-air reconfiguration of remote software radio terminals using the free GNU Radio toolkit, facilitating the development of Cognitive Radio applications. The modular architecture allows for separating the cognitive engine from the signal processing algorithms, thus allowing fast implementation, verification and easier testing.**

***Index Terms*—Over-The-Air (OTA) Reconfiguration, Cognitive Radio, GNU Radio**

I. Introduction

Cognitive Radio (CR) is a broad term and entails a variety of concepts and technologies. In this work we attempt to tackle one of the problems often neglected in this context: the remote reconfiguration of radio terminals (*over-the-air reconfiguration*). This element is necessary to benefit from the decisions of a cognitive engine (CE): if a terminal dynamically reconfigures the waveform applied, it must be possible to transmit this information to the other terminals. Over-the-air (OTA) reconfiguration is described in [1].

Our work consists of a software package able to drive Software Defined Radio (SDR) hardware for CR applications. We provide a minimal framework which facilitates the remote reconfiguration of other terminals. By this means, it could be possible to attach a CE or other elements such as a spectral observation module which in turn select the applicable waveform for a given scenario.

II. Hardware/Software

We use the free software radio toolkit GNU Radio [2] for implementation. Every terminal in the wireless network consists of a host PC running the same version of GNU Radio. It is possible to include external GNU Radio modules such as the spectral estimation toolbox [3] or any other kind of software module; however, it is assumed a-priori that the base system is identical in all terminals to ensure that software running on one terminal will also run on another.

Every terminal host PC is connected to an identical radio front-end such as the Universal Software Radio Peripheral (USRP). Identical in this context refers to the fact that all the RF front-ends have equal capabilities (such as bandwidth, frequency range) and expose the same application programming interface (API).

The combination of GNU Radio and the USRP allows for very small executable files while at the same time allowing for simple and fast development of signal processing code. Using GNU Radio, it is possible to write powerful software radio code in a single file of a few kilobytes in size, especially if compression is applied to the Python source code.

We combine our software framework in a package called *GROTARAPI* (*GNU Radio over the air reconfiguration API*).

III. Capabilities

The purpose of this experimental software is to provide a testbed for CR algorithms. The main feature of such software must thus be modularity: It is essential that central parts of the software can quickly and easily be exchanged. Our design therefore separates signal processing components from the decision logic, but provides easy access to the individual components.

A. *Remote over-the-air reconfiguration*

The main feature is the built-in capability to reconfigure remote terminals via the air interface. This is done by transmitting the actual signal processing modules themselves. If one terminal – denoted “master” here – wants to initiate a communication by a given standard, a request is transmitted to the receiving – or “slave” – terminal via the control channel. If the receiving terminal does not know this specific standard, the master terminal transmits the code necessary. Once both terminals have the signal processing code necessary, both terminals activate this waveform and use it for ensuing communications.

B. *Security issues*

Since the master terminal transmits code that is directly executed on the slave terminal, a few words on security are in order. We apply a GPG signature [4] to transmitted code for the receiver to be able to verify the true identity of the transmitter. Still, this kind of setup implements only rudimentary security. Direct execution of code which must be able to access connected hardware is a vulnerability. It must be ensured that experiments are run in a controlled environment.

IV. Implementation Details

As mentioned before, we solely focus on the OTA reconfiguration problem as described in Section III, while exposing an interface to add CR features later on. Careful investigation of the required capabilities of a cognitive radio platform as described in [5] show that access to a wide range of meters such as battery status or geographic location are needed in

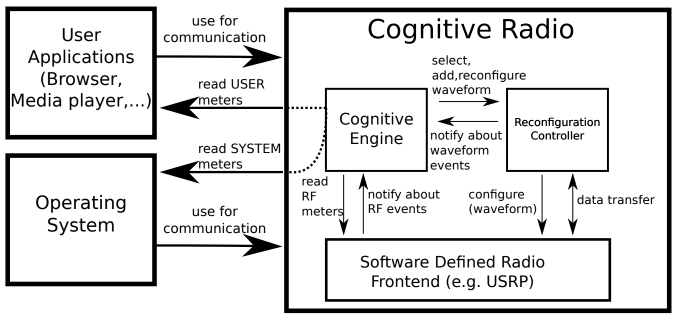


Fig. 1. Global Overview

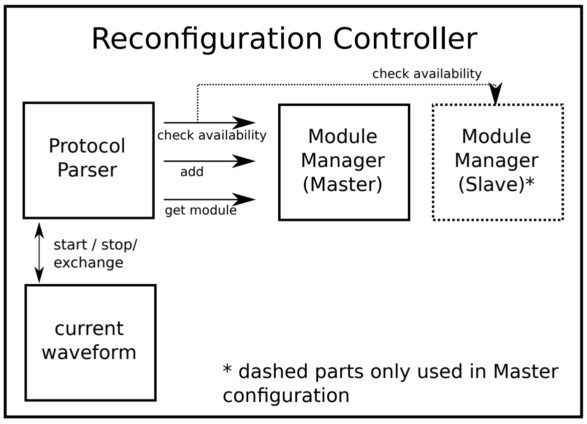


Fig. 2. Overview of the Reconfiguration Controller

order for the CR being able to make "smart" decisions. It is thus sensible to provide the interface to the other parts of the CR such as the CE in a way that also facilitates access to the meter values, while still staying flexible enough to not put unnecessary restrictions for the overall design in place.

A. Overview

Fig. 1 gives a rough overview on how to integrate the CR's cognitive engine, user applications and our Reconfiguration Controller. The CR is composed of an SDR front-end, a Cognitive Engine, and our Reconfiguration Controller. In order to work out of the box with our framework, the SDR front-end needs to fulfill the requirements outlined in Section I.

The *Cognitive Engine* (CE) can be considered the "brains" of a Cognitive Radio. Using adaptive algorithms, machine learning and artificial intelligence techniques (see e.g. [6]) to evaluate the *meter* values it actively influences the behaviour of the CR system ("turning the knobs"). However, this problem is not subject of our implementation.

The *Reconfiguration Controller* (RC) allows the cognitive engine to *add*, *reconfigure* or *select* a waveform via a simple remote procedure call (RPC). It provides an interface via D-Bus (Section IV-B). Additionally, it is possible for the cognitive engine to register for notification about certain events such as the successful switch from one waveform to another, or the availability of a new waveform. This facilitates asynchronous programming.

By starting a certain waveform the Reconfiguration Controller reconfigures the SDR front-end to communicate conforming to a certain standard. Furthermore, the Reconfiguration Controller enables a secure and reliable waveform exchange between the terminals.

Fig. 2 gives an overview over the parts of the Reconfiguration Controller. In the master configuration it is composed of a *Protocol Parser*, two *Module Managers* and a reference to the current waveform.

The waveforms in our implementation are composed of building blocks we called *modules*. These *modules* can be Python source code files, binary files, or arbitrary other files.

In order to manage availability, security, and deployment of the modules we use an entity termed the *Module Manager*. The *Module Manager* currently allows for getting a list of available modules, adding a new module, verifying the signature of a module and compiling a list of missing modules, if given a list of desired modules.

In addition to the Module Manager the Reconfiguration Controller contains a *Protocol Parser*. This parser consists of a state machine that represents the protocol (See Section IV-D) The Protocol Parser (and thereby the used protocol) can be easily exchanged to use e.g. a binary protocol if performance is an issue. Moreover, small adaptions can be easily made, to cope with smaller changes, such as adding custom commands in case the CE needs more elaborate commands for the Reconfiguration Controller.

Let us consider the following example to illustrate how the parts of the CR might work together:

1) By constantly monitoring the RF spectrum, the cognitive engine creates a new waveform, to use a newly found spectrum hole for communication.
2) After creating the waveform the cognitive engine notifies the Reconfiguration Controller about the availability of a new waveform and requests to run this new waveform.
3) The Reconfiguration Controller will ensure the availability of the waveform for all participating terminals.
4) Finally both terminals switch to the new waveform, and the Reconfiguration Controller notifies the cognitive engine about the successful change.

B. The D-Bus interface

In order to enhance flexibility when designing CRs using our software, it is possible to run the different parts of the CR in separate processes; however, this creates the need for inter process communication (IPC). The D-Bus framework[1] was chosen instead of traditional UNIX methods such as pipes or shared memory for it's ease of use and great flexibility. By registering a service with D-Bus' session bus it is easily possible for other parts of the CR to access the Reconfiguration Controller's capabilities. Using D-Bus, the task of collecting

[1]Desktop BUS is part of the freedesktop.org project, see [7] for further reference.

the meter's values from different sources in the system and keeping the different parts of the cognitive radio updated about events, are reduced to simple RPCs or registering handlers for the signals emitted by other parts of the CR.

A simple battery meter to show how easy it is to add custom meters to the cognitive radio system using D-Bus has been implemented. Via a simple RPC it is possible to get the current battery discharge rate or the remaining capacity. In order to save resources (processor load), the distributed meter concept could be further enhanced to use D-Bus' ability to transparently start services when a RPC is made and they are not running.

C. Control Channels

Since the individual terminals can only communicate via air interface (which spawns the need for OTA reconfiguration in the first place), we require some kind of control channel to transmit protocol data and signal processing from one terminal to the other. This is a simple task within the presented setup, since the reconfiguration controller has access to the entire library of waveform modules, which it can use to transmit commands to the remote RCs.

This leaves the problem of bootstrapping the communication. At one point in time, all RCs must know where to "listen" for incoming commands. In our setup, we define a *Control Channel* by choosing a fixed frequency and waveform. At the beginning of a reconfiguration cycle, the terminals switch to this control channel and wait for or transmit commands to configure the ensuing communication. This step is called the *rendezvous period*. However, alternatives to a fixed control channel have been proposed previously (e.g. [8, Section V]). Also, the protocol can be extended to change the control channel between reconfigurations.

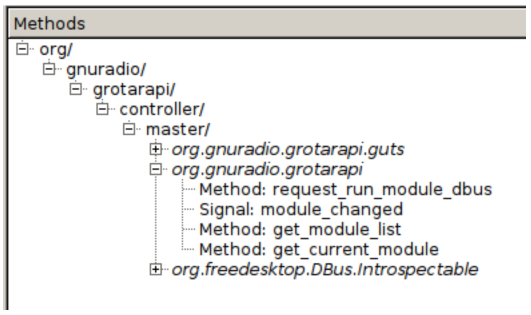
D. The OTA Reconfiguration Protocol

After the initial rendezvous, the terminals start normal communication (*idle mode*). For simplicity and readability reasons we used a simple line based protocol. Requests to change the waveform for example have the following form:

```
RUN_MOD
{NAME_OF_THE_WAVEFORM}
ARGS
{STRING_OF_ARGUMENTS}
```

The arguments contain information such as the centre frequency or the maximum time after which the waveform should be switched back to the control channel. We simply store these as a set of key/value mappings. Python has built-in libraries to serialise this kind of data into strings, which we utilise to efficiently and simply transmit the arguments in string form within the protocol stream.

Fig. 3 shows an example on how the OTA reconfiguration protocol works. For this setup, the CE is replaced by users who explicitly request waveforms from the radio systems. We have two waveforms available (A and B), which differ in modulation, bandwidth and center frequency. Also, we fix a physical channel, which is known to both terminals, as a control channel.



```
Methods
⊟ org/
  ⊟ gnuradio/
    ⊟ grotarapi/
      ⊟ controller/
        ⊟ master/
          ⊞ org.gnuradio.grotarapi.guts
          ⊟ org.gnuradio.grotarapi
              Method: request_run_module_dbus
              Signal: module_changed
              Method: get_module_list
              Method: get_current_module
          ⊞ org.freedesktop.DBus.Introspectable
```

Fig. 4. Screenshot of QDBusViewer showing the master's D-Bus interface

In the first step, one user requests to start a transmission using waveform A. The RC checks this waveform is actually available and transmits the command `RUN_MOD {waveform A}` to the other terminal. Here, again, the RC checks if this waveform is available. This is the case, so it replies with a `RUN_ACK` acknowledgement, and both terminals reconfigure themselves to use the requested waveform. Assuming both terminals only have one RF front-end, the hardware is then blocked by waveform A until the waveform process exits; either by a given timeout or simply by exiting itself (e.g. after a successful data transmission).

After exiting, the RC takes control of the SDR again. The user can then request communication using a different waveform and selects waveform B. Again, the RC of the first terminal requests communication from the other terminal using the command `RUN_MOD {waveform B}`. This time, the remote terminal does not have the waveform available and replies with a negative acknowledgement, `RUN_NAK`. The first RC then transmits the communication modules to other terminal, which checks signature to ensure no invalid code was introduced and adds the checked code into its own module library. Once this has happened, it can transmit the *Run Acknowledge* `RUN_ACK`. Both terminals can then communicate using waveform B.

E. Experimental Setup

All experiments were conducted using two off-the-shelf laptops running a recent Linux distribution. Laptop A was running a 64-Bit Linux for Intel's em64t architecture. Laptop B was running a 32-bit Linux for Intel's x86 architecture; however, this does not mean that the implementation is limited to x86 processors. Any architecture capable of running GNU Radio and Python in the required versions (GNU Radio 3.4 compiled with UHD support, Python 2.7) should be able to drive the proposed platform. As RF front-end two USRP2 devices using Wideband Transceiver (WBX) daughterboards were used. The WBX boards allow transmission and reception in a frequency range from 50 MHz to 2.2 GHz, thus allowing for great flexibility. The USRP2s were connected to the host laptops via gigabit Ethernet, allowing for up to 20 MHz of usable RF bandwidth, limited by the host PC's ability to

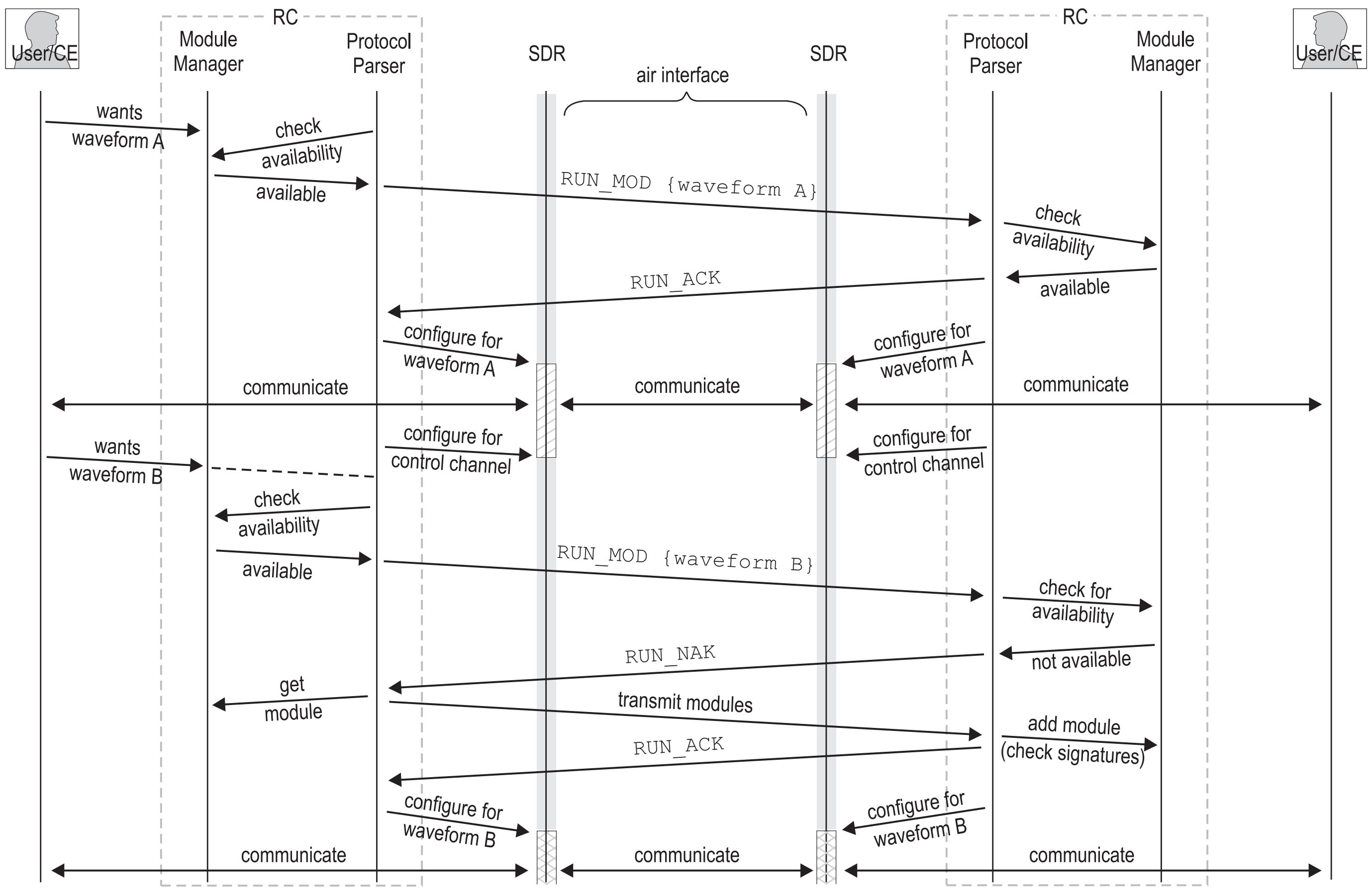


Fig. 3. Example setup with protocol details

process this amount of data and, of course, the employed waveform. We successfully implemented a demonstration of the testbed's capabilities by implementing three exemplary waveforms using the GNU Radio SDR framework: A video and an audio stream using GMSK modulation as well as an analog narrowband FM waveform.

As mentioned in Section IV-A, the Reconfiguration Controller exposes an interface that can be explored using graphical tools such as *QDBusViewer*, a graphical D-Bus exploration tool [9].

Fig. 4 shows the exposed signal *module_changed* that will be emitted if the waveform is finally changed to the one requested by the RPC call *request_run_module_dbus* that might be used by a CR designer working on the CE to easily initiate the OTA reconfiguration of a second terminal.

V. Conclusion

In this paper, we introduce a testbed for the implementation of CR algorithms using Python and the GNU Radio framework. The proposed Reconfiguration Controller takes care of the OTA reconfiguration which is required in a CR system to allow the designer to concentrate on the implementation of CR algorithms and the details of the Cognitive Engine. It can be regarded as one of the many necessary steps to implement a CR system as envisioned in [5].

As Section IV-B shows, D-Bus appears to be quite useful in a PC-based CR architecture for its ease of use and high flexibility.

References

[1] M. Jones and L. McGrath, "Over-the-air software download considerations for public safety and other markets," *Proceedings of the SDR '05 Technical Conference*, 2005.

[2] GNU Radio Project Website. [Online]. Available: www.gnuradio.org

[3] *SpecEst: The Spectral Estimation Toolbox for GNU Radio*, Communications Engineering Lab, Karlsruhe Institute of Technology, 2010. [Online]. Available: http://www.cgran.org/wiki/SpecEst

[4] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880 (Proposed Standard), Internet Engineering Task Force, Nov. 2007, updated by RFC 5581. [Online]. Available: http://www.ietf.org/rfc/rfc4880.txt

[5] J. Mitola, "Cognitive Radio – An Integrated Agent Architecture for Software Defined Radio," Ph.D. dissertation, Royal Institute of Technology (KTH), Sweden, 2000.

[6] T. W. Rondeau and C. W. Bostian, *Artificial intelligence in wireless communications*, ser. Mobile Communications Series. Boston [u.a.]: Artech House, 2009.

[7] Freedesktop.org D-BUS Project Website. [Online]. Available: www.freedesktop.org/software/dbus

[8] M. Braun, J. P. Elsner, and F. K. Jondral, "Signal Detection for Cognitive Radios with Smashed Filtering," *IEEE Vehicular Technology Conference*, April 2009.

[9] QDBusViewer Online Reference. [Online]. Available: http://doc.qt.nokia.com/4.5/qdbusviewer.html