Dept. of Electrical Engineering

IIT Bombay

Dual Degree Project Stage 1

OpenBTS with cognitive capabilities

Authors:
Swrangsar Basumatary
Abrar Ahmad

Supervisor:
Prof. S N Merchant

September 20, 2013

Abstract

Our goal is to set up an OpenBTS system with cognitive capabilities. We have a predefined frequency band to run our cognitive OpenBTS system in. First we sense the presence of primary users in that particular frequency band by detecting the presence of ongoing calls. If it turns out that the calls made by primary users end, then we start our secondary OpenBTS system thus allowing secondary users to make calls.

Contents

Chapter 1

Introduction

1.1 Cognitive Radio

A cognitive radio is an intelligent radio that can be programmed and configured dynamically. Its transceiver is designed to use the best wireless channels in its vicinity. Such a radio automatically detects available channels in wireless spectrum, then accordingly changes its transmission or reception parameters to allow more concurrent wireless communications in a given spectrum band at one location. This process is a form of dynamic spectrum management[1].

1.2 Motivation

Studies have shown that most of the spectrum allotted to licensed networks remain unused most of the time[2]. To utilize these unused spectral resources we can make use of dynamic spectrum management. We can allow secondary (unlicensed) users to utilize the spectrum whenever that particular spectrum becomes available. For this we need cognitive capabilities to sense the availability of the spectrum.

1.3 Organization

Chapter 2

Software Defined Radio

2.1 Introduction

2.2 USRP

USRP (Universal Software Radio Peripheral) is a hardware kit developed by Ettus, Inc. to run software defined radio applications.

2.3 GnuRadio

2.4 OpenBTS

Chapter 3

Spectrum Sensing

3.1 Matched filter detection

3.2 Energy based detection

Chapter 4

A cognitive base station using GnuRadio and OpenBTS

4.1 Motivation

Appendix A

Codes

A.1 senseUplinknStartBTS.py

```
#!/usr/bin/env python
#
# Copyright 2005,2007,2011 Free Software Foundation,
    Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it
    and/or modify
# it under the terms of the GNU General Public License
    as published by
# the Free Software Foundation; either version 3, or (
    at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be
    useful,
# but WITHOUT ANY WARRANTY; without even the implied
    warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
     See the
# GNU General Public License for more details.
#
```

```
# You should have received a copy of the GNU General
    Public License
# along with GNU Radio; see the file COPYING.  If not,
    write to
# the Free Software Foundation, Inc., 51 Franklin
    Street,
# Boston, MA 02110-1301, USA.
#

from gnuradio import gr, eng_notation
from gnuradio import blocks
from gnuradio import audio
from gnuradio import filter
from gnuradio import fft
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from optparse import OptionParser
import sys
import math
import struct
import threading
import time
import sqlite3
import os
import subprocess
from datetime import datetime

sys.stderr.write("Warning: this may have issues on some
    machines+Python version combinations to seg fault
    due to the callback in bin_statitics.\n\n")

class ThreadClass(threading.Thread):
    def run(self):
        return

class tune(gr.feval_dd):
    """
    This class allows C++ code to callback into python.
    """
    def __init__(self, tb):
        gr.feval_dd.__init__(self)
```

```
        self.tb = tb

    def eval(self, ignore):
        """
        This method is called from blocks.bin_statistics_f when it wants
        to change the center frequency.  This method tunes the front
        end to the new center frequency, and returns the new frequency
        as its result.
        """

        try:
            # We use this try block so that if something goes wrong
            # from here down, at least we'll have a prayer of knowing
            # what went wrong.  Without this, you get a very
            # mysterious:
            #
            #   terminate called after throwing an instance of
            #   'Swig::DirectorMethodException' Aborted
            #
            # message on stderr.  Not exactly helpful ;)

            new_freq = self.tb.set_next_freq()

            # wait until msgq is empty before continuing
            while(self.tb.msgq.full_p()):
                #print "msgq full, holding.."
                time.sleep(0.1)

            return new_freq

        except Exception, e:
            print "tune: Exception: ", e
```

```
class parse_msg(object):
    def __init__(self, msg):
        self.center_freq = msg.arg1()
        self.vlen = int(msg.arg2())
        assert(msg.length() == self.vlen * gr.
            sizeof_float)

        # FIXME consider using NumPy array
        t = msg.to_string()
        self.raw_data = t
        self.data = struct.unpack('%df' % (self.vlen,),
            t)


class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

        usage = "usage: %prog [options] down_freq"
        parser = OptionParser(option_class=eng_option,
            usage=usage)
        parser.add_option("-a", "--args", type="string"
            , default="",
                          help="UHD device device
                              address args [default=%
                              default]")
        parser.add_option("", "--spec", type="string",
            default=None,
                          help="Subdevice of UHD device
                              where appropriate")
        parser.add_option("-A", "--antenna", type="
            string", default=None,
                          help="select Rx Antenna where
                              appropriate")
        parser.add_option("-s", "--samp-rate", type="
            eng_float", default=10e6,
                          help="set sample rate [
                              default=%default]")
```

```
parser.add_option("-g", "--gain", type="
    eng_float", default=None,
                      help="set gain in dB (default
                          is midpoint)")
parser.add_option("", "--tune-delay", type="
    eng_float",
                      default=0.25, metavar="SECS",
                      help="time to delay (in
                          seconds) after changing
                          frequency [default=%
                          default]")
parser.add_option("", "--dwell-delay", type="
    eng_float",
                      default=0.25, metavar="SECS",
                      help="time to dwell (in
                          seconds) at a given
                          frequency [default=%
                          default]")
parser.add_option("-b", "--channel-bandwidth",
    type="eng_float",
                      default=9.7656e3, metavar="Hz
                          ",
                      help="channel bandwidth of
                          fft bins in Hz [default=%
                          default]")
parser.add_option("-l", "--lo-offset", type="
    eng_float",
                      default=0, metavar="Hz",
                      help="lo_offset in Hz [
                          default=%default]")
parser.add_option("-q", "--squelch-threshold",
    type="eng_float",
                      default=None, metavar="dB",
                      help="squelch threshold in dB
                          [default=%default]")
parser.add_option("-F", "--fft-size", type="int
    ", default=None,
                      help="specify number of FFT
                          bins [default=samp_rate/
                          channel_bw]")
```

```
parser.add_option("", "--real-time", action="
    store_true", default=False,
                      help="Attempt to enable real-
                          time scheduling")


(options, args) = parser.parse_args()
if len(args) != 1:
    parser.print_help()
    sys.exit(1)

self.channel_bandwidth = options.
    channel_bandwidth

self.down_freq = eng_notation.str_to_num(args
    [0])
self.up_freq = self.down_freq - 45e6



if not options.real_time:
    realtime = False
else:
    # Attempt to enable realtime scheduling
    r = gr.enable_realtime_scheduling()
    if r == gr.RT_OK:
        realtime = True
    else:
        realtime = False
        print "Note: failed to enable realtime
            scheduling"

# build graph
self.u = uhd.usrp_source(device_addr=options.
    args,
                             stream_args=uhd.
                                 stream_args('fc32')
                                 )

# Set the subdevice spec
if(options.spec):
```

```
        self.u.set_subdev_spec(options.spec, 0)

    # Set the antenna
    if(options.antenna):
        self.u.set_antenna(options.antenna, 0)

    self.u.set_samp_rate(options.samp_rate)

    self.usrp_rate = usrp_rate = self.u.
       get_samp_rate()

    self.lo_offset = options.lo_offset

    if options.fft_size is None:
        self.fft_size = int(self.usrp_rate/self.
           channel_bandwidth)
    else:
        self.fft_size = options.fft_size

    self.squelch_threshold = options.
       squelch_threshold

    s2v = blocks.stream_to_vector(gr.
       sizeof_gr_complex, self.fft_size)

    mywindow = filter.window.blackmanharris(self.
       fft_size)
    ffter = fft.fft_vcc(self.fft_size, True,
       mywindow, True)
    power = 0
    for tap in mywindow:
        power += tap*tap

    c2mag = blocks.complex_to_mag_squared(self.
       fft_size)


    tune_delay  = max(0, int(round(options.
       tune_delay * usrp_rate / self.fft_size))) #
        in fft_frames
```

```
        dwell_delay = max(1, int(round(options.
            dwell_delay * usrp_rate / self.fft_size))) #
             in fft_frames

        self.msgq = gr.msg_queue(1)
        self._tune_callback = tune(self)        # hang
            on to this to keep it from being GC'd
        stats = blocks.bin_statistics_f(self.fft_size,
            self.msgq,
                                        self.
                                            _tune_callback
                                            , tune_delay
                                            ,
                                        dwell_delay)

        # FIXME leave out the log10 until we speed it
            up
        #self.connect(self.u, s2v, ffter, c2mag, log,
            stats)
        self.connect(self.u, s2v, ffter, c2mag, stats)

        if options.gain is None:
            # if no gain was specified, use the mid-
                point in dB
            g = self.u.get_gain_range()
            options.gain = float(g.start()+g.stop())
                /2.0

        self.set_gain(options.gain)
        print "gain =", options.gain

    def set_next_freq(self):
        target_freq = self.up_freq


        if not self.set_freq(target_freq):
            print "Failed to set frequency to",
                target_freq
            sys.exit(1)

        return target_freq
```

```
    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        Args:
            target_freq: frequency in Hz
        @rypte: bool
        """

        r = self.u.set_center_freq(uhd.tune_request(
            target_freq, rf_freq=(target_freq + self.
            lo_offset),rf_freq_policy=uhd.tune_request.
            POLICY_MANUAL))
        if r:
            return True

        return False

    def set_gain(self, gain):
        self.u.set_gain(gain)



def main_loop(tb):

    # use a counter to make sure power is less than
        threshold
    lowPowerCount = 0
    lowPowerCountMax = 10
    print 'fft size', tb.fft_size
    N = tb.fft_size


    while 1:

        # Get the next message sent from the C++ code (
            blocking call).
        # It contains the center frequency and the mag
            squared of the fft
```

```
            m = parse_msg(tb.msgq.delete_head())

            # m.center_freq is the center frequency at the
                time of capture
            # m.data are the mag_squared of the fft output
            # m.raw_data is a string that contains the
                binary floats.
            # You could write this as binary to a file.



            center_freq = m.center_freq
            bins = 10
            power_data = 0

            for i in range(1, bins+1):
                power_data += m.data[N-i] + m.data[i]
            power_data += m.data[0]
            power_data /= ((2*bins) + 1)

            power_db = 10*math.log10(power_data/tb.
                usrp_rate)
            power_threshold = -95

            if (power_db > tb.squelch_threshold) and (
                power_db > power_threshold):
                print datetime.now(), "center_freq",
                    center_freq, "power_db", power_db, "in
                    use"
                lowPowerCount = 0
            else:
                print datetime.now(), "center_freq",
                    center_freq, "power_db", power_db
                lowPowerCount += 1
                if (lowPowerCount > lowPowerCountMax):
                    down_freq = center_freq + 45e6
                    startOpenBTS(down_freq)
                    break


def startOpenBTS(downFrequency):
```

```
    arfcn=int((downFrequency-935e6)/2e5)
    if (arfcn < 0):
        print "ARFCN must be > 0 !!!"
        sys.exit(1)
    print 'ARFCN=', arfcn
    #DB modifications
    t=(arfcn,)
    conn=sqlite3.connect("/etc/OpenBTS/OpenBTS.db")
    cursor=conn.cursor()
    cursor.execute("update config set valuestring=?
       where keystring='GSM.Radio.C0'",t)
    conn.commit()

    #start the OpenBTS
    f=subprocess.Popen(os.path.expanduser('~/ddp-stage
       -1-and-openbts/runOpenBTS.sh'))
    f.wait()


if __name__ == '__main__':
    t = ThreadClass()
    t.start()

    tb = my_top_block()
    try:
        tb.start()
        main_loop(tb)

    except KeyboardInterrupt:
        pass
```

Bibliography

[1] `http://en.wikipedia.org/wiki/Cognitive_radio`.

[2] Federal Communications Commission. Spectrum policy task force. *ET Docket No. 02-135*, November 2002.