GNURadio Testbed

Naveen Manicka
(manicka@udel.edu)

Advisor – Dr. Chien-Chung Shen
Committee Member – Dr. Stephan Bohacek

Dept of Computer and Information Sciences
University of Delaware
Thesis overview

- Tutorial
  - SDR overview
  - Gnuradio installation guide
  - Gnuradio internals

- Network Testbed
  - MAC protocol implementation framework
  - Processing blocks data capture and plotting in Matlab
Thesis overview

- Tutorial
  - SDR overview
  - Gnuradio installation guide
  - Gnuradio internals

- Network Testbed
  - MAC protocol implementation framework
  - Processing blocks data capture and plotting in Matlab
Outline

- Hardware Radio Shortcomings
- Software Defined Radio
- GNU Software Radio
Shortcomings of present day radios

- Performance & functionality fixed from design stage
- Not much room for enhancement
- Constant evolution of wireless link layer protocols
- Incompatible wireless network technologies, inhibiting global roaming
Software Defined Radio – SDR

- Implement radio functionality as software modules running on a general purpose basic hardware

- Can be completely redefined in the field through modification of software

- Advantages
  
  - Switch between different architectures (implemented as software modules)
  
  - Significant improvement in price/performance over traditional radio
    
    - Over the air deployment of software modules to subscribers
    
    - Adapt to changing network requirements
**SDR features**

- Re-configurable
  - Co-existence of multiple standards and infrastructure equipment

- Ubiquitous connectivity
  - Global roaming
  - Over the air upgradation

- Open architecture – drivers are OS dependant

- Enhanced utility
Generic SDR architecture

- Higher Layer Protocols (WAP, TCP/IP)
- Radio Applications (Modulation, Demodulation)
- Hardware Resources (USRP)
Generic digital transceiver
Analog & digital signals

- Nyquist criterion: $f_s \geq 2f_{bw}$
- Audible range: 20 Hz – 20 KHz
  - Required sampling $\geq 40$ KHz
  - Ex: CD player std sampling rate = 44.1 KHz
From RF to PC soundcard

- Convert modulated RF signal from Frequency domain to Time domain
  - **Frequency domain**: Amplitude vs Frequency, as measured by a spectrum analyzer
  - **Time domain**: Amplitude vs Time, as measured by an oscilloscope

- Standard 16 bit PC soundcard -> Max sampling rate is 44.1KHz

- Conversion from RF to Baseband: Mix RF signal with oscillator signal
  - \( f_c f_{lo} \rightarrow f_c + f_{lo} \) & \( f_c - f_{lo} \) and image signals \(-f_c + f_{lo}\) & \(-f_c - f_{lo}\)
Quadrature mixing & FFT

Low pass filter removes sum signal and its image
Difference signal image removed using quadrature mixing
FFT converts I & Q into frequency domain
Inverse FFT converts signal back to time domain

Instantaneous Magnitude

\[ m_t = \sqrt{I_t^2 + Q_t^2} \]

Instantaneous Phase

\[ \phi_t = \tan^{-1}\left(\frac{Q_t}{I_t}\right) \]
GNU Radio

- Software Structure
**USRP - Universal Software Radio Peripheral**
GNU Radio

- Library of signal processing blocks written in C++
  - Signal sources
  - Signal sinks
  - Filters

- Glued together using Python

- SDR -> Create a graph, with vertices as signal processing blocks and edges as data flow between vertices.

- Block attributes defines the # of input & output ports and type of data flow through them
Available GNU radio blocks
(Partial list)

- **Signal Sources**
  - Sinusoidal
  - Noise
  - Audio
  - USRP

- **Signal Sinks**
  - Null
  - File
  - Audio
  - USRP

- **Simple Operators**
  - Adder
  - Subtractor
  - Multiplier

- **Filters**
  - FIR
  - IIR

- **Other useful blocks**
  - FM modulation and demodulation
  - Blocks for digital transmission

Etc.....
Basic model

Packet Transmitter

Noise

ADDERR

Receiver Filter

Demodulator

Signal Source

Modulator

Packet Transmitter
GNU Radio – Hello World!

Dial Tone Output

#!/usr/bin/env python
from gnuradio import gr
from gnuradio import audio

def build_graph ():
    sampling_freq = 48000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect ((src0, 0), (dst, 0))
    fg.connect ((src1, 0), (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
TUN/TAP adapters

- TUN/TAP provides packet reception and transmission for user space programs
- Reads/writes packets from/to the user space program instead of receiving/sending packets via physical media
- TUN works with IP frames. TAP works with Ethernet frames.
- In order to use the driver a program has to open /dev/net/tun and issue a corresponding ioctl() to register a network device with the kernel. A network device will appear as tunXX or tapXX, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear. Depending on the type of device chosen the userspace program has to read/write IP packets (with tun) or ethernet frames (with tap).
Data flow - Send path

Application (For Ex. PING) → Kernel → TUN Interface → GNU Radio Software (MAC Implementation) → USRP
Data flow - Receive path

Application (For Ex. PING) → Kernel → TUN Interface → GNU Radio Software (MAC Implementation) → USRP
Data communication path

- Application (For Ex. PING)
- Kernel
- TUN Interface
- GNU Radio Software (MAC Implementation)
- USRP

- Application (For Ex. PING)
- Kernel
- TUN Interface
- GNU Radio Software (MAC Implementation)
- USRP
Tunnel.py

```python
# ///////////////////////////////////////////////////////////////////////////////////
#
# Use the Universal TUN/TAP device driver to move packets to/from kernel
#
# ///////////////////////////////////////////////////////////////////////////////////

tun = os.open(tun_device_filename, os.O_RDWR)
ifs = ioctl(tun, TUNSETIFF, struct.pack("16sH", "gr%d", mode))
ifname = ifs[:16].strip("\x00")
return (tun, ifname)

# ///////////////////////////////////////////////////////////////////////////////////
#    the flow graph
# ///////////////////////////////////////////////////////////////////////////////////

class my_graph(gr.flow_graph):

    def __init__(self, mod_class, demod_class,
                 rx_callback, options):
        gr.flow_graph.__init__(self)
        self.txpath = transmit_path(self, mod_class, options)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)

    def send_pkt(self, payload="", eof=False):
        return self.txpath.send_pkt(payload, eof)

    def carrier_sensed(self):
        return True if self.rxpath.carrier_sensed()

# /////////////////////////////////////////////////////////////////////////////
#                                   main
# /////////////////////////////////////////////////////////////////////////////

def main():
    # instantiate the MAC
    mac = cs_mac(tun_fd, verbose=True)

    # build the graph (PHY)
    fg = my_graph(mods[options.modulation],
                  demods[options.modulation],
                  mac.phy_rx_callback,
                  options)
    mac.set_flow_graph(fg)  # give the MAC a handle for the PHY

    if fg.txpath.bitrate() != fg.rxpath.bitrate():
        print "WARNING: Transmit bitrate = %sb/sec, Receive bitrate = %sb/sec" % (eng_notation.num_to_str(fg.txpath.bitrate()), eng_notation.num_to_str(fg.rxpath.bitrate()))
        print "modulation:     %s"   % (options.modulation,)  
        print "freq:           %s"      % (eng_notation.num_to_str(options.tx_freq)) 
        print "bitrate:        %sb/sec" % (eng_notation.num_to_str(fg.txpath.bitrate())) 
        print "samples/symbol: %3d" % (fg.txpath.samples_per_symbol(),) 
        print "interp:         %3d" % (fg.txpath.interp(),) 
        print "decim:          %3d" % (fg.txpath.decim(),)
        fg.rxpath.set_carrier_threshold(options.carrier_threshold)
        print "Carrier sense threshold: ", options.carrier_threshold, "dB"
```
Thesis overview

- Tutorial
  - SDR overview
  - Gnuradio installation guide
  - Gnuradio internals

- Network Testbed
  - MAC protocol implementation framework
  - Processing blocks data capture and plotting in Matlab
MAC functions

Qualnet

- NetworkLayerHasPacketToSend
- ReceivePacketFromPhy
- ReceivePhyStatusChangeNotification
- MessageSend()
- Message_Alloc

Gnuradio

- os.read – reads head from queue
- phy_rx_callback()
- carrier_sensed()
- Send_pkt()
- Make_pkt()
Tunnel.py

# ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
# # Use the Universal TUN/TAP device driver to move packets to/from kernel
# # //_/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

import os
import struct

# ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
tun = os.open(tun_device_filename, os.O_RDWR)
ifs = ioctl(tun, TUNSETIFF, struct.pack("16sH", "gr%d", mode))
ifname = ifs[:16].strip("\00")
return (tun, ifname)

# //_/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
# # the flow graph
# //_/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class my_graph(gr.flow_graph):
    def __init__(self, mod_class, demod_class, rx_callback, options):
        gr.flow_graph.__init__(self)
        self.txpath = transmit_path(self, mod_class, options)
        self.rxpath = receive_path(self, demod_class, rx_callback, options)

        def send_pkt(self, payload='', eof=False):
            return self.txpath.send_pkt(payload, eof)

        def carrier_sensed(self):
            return self.rxpath.carrier_sensed()

    def main():
        """# instantiate the MAC
        mac = cs_mac(tun_fd, verbose=True)
        
        # build the graph (PHY)
        fg = my_graph(mods[options.modulation],
                       demods[options.modulation],
                       mac.phy_rx_callback,
                       options)
        mac.set_flow_graph(fg)  # give the MAC a handle for the PHY
        if fg.txpath.bitrate() != fg.rxpath.bitrate():
            print("WARNING: Transmit bitrate = %s, Receive bitrate = %s")
            print(eng_notation.num_to_str(fg.txpath.bitrate()),
                  eng_notation.num_to_str(fg.rxpath.bitrate()))
            print("modulation: %s", options.modulation,)
            print("freq: %s", options.tx_freq)
            print("bitrate: %s", options.bitrate)
            print("samples/symbol: %s", options.samples_per_symbol())
            print("interp: %s", options.interp())
            print("decim: %s", options.decim())
            print("Carrier sense threshold: ", options.carrier_threshold, "dB")

        fg.rxpath.set_carrier_threshold(options.carrier_threshold)
        print("Carrier sense threshold: ", options.carrier_threshold, "dB")"""
PHY receives packet

def phy_rx_callback(self, ok, payload):
    if self.verbose:
        print "Rx: ok = %r  len(payload) = %4d" % (ok, len(payload))
    if ok:
        os.write(self.tun_fd, payload)
Network layer has packet to send

def main_loop(self):
    min_delay = 0.001  # seconds
    while 1:
        payload = os.read(self.tun_fd, 10*1024)

        if not payload:
            self.fg.send_pkt(eof=True)
            break

        if self.verbose:
            print "Tx: len(payload) = %4d" % (len(payload),)

        delay = min_delay

        while self.fg.carrier_sensed():
            sys.stderr.write('B'
            time.sleep(delay)
            if delay < 0.050:
                delay = delay * 2  # exponential back-off

        self.fg.send_pkt(payload)
Modulating and demodulating packets

**Mod_pkt:**
- Hierarchical block for sending packets

Packets to be sent are enqueued by calling send_pkt. The output is the complex modulated signal at baseband.

@param fg: flow graph

@param modulator: instance of modulator class (gr_block or hier_block)

@param access_code: AKA sync vector
@type access_code: string of 1’s and 0’s between 1 and 64 long

@param msgq_limit: maximum number of messages in message queue

@param pad_for_usrp: If true, packets are padded such that they end up a multiple of 128 samples

**Demod_pkt:**
- Hierarchical block for demodulating and deframing packets.

The input is the complex modulated signal at baseband. Demodulated packets are sent to the handler.

@param fg: flow graph

@param demodulator: instance of demodulator class (gr_block or hier_block)
@type demodulator: complex baseband in

@param access_code: AKA sync vector
@type access_code: string of 1’s and 0’s

@param callback: function of two args: ok, payload
@type callback: ok: bool; payload: string

@param threshold: detect access_code with up to threshold bits wrong (-1 -> use default)
Packet format

Make_pkt:
- Build a packet, given access code and payload.

@param payload: packet payload, len [0, 4096]
@param samples_per_symbol: samples per symbol (needed for padding calculation)
@type samples_per_symbol: int
@param bits_per_symbol: (needed for padding calculation)
@type bits_per_symbol: int
@param access_code: string of ascii 0's and 1's

Padding:
- Generate sufficient padding such that each packet ultimately ends up being a multiple of 512 bytes when sent across the USB. We send 4-byte samples across the USB (16-bit I and 16-bit Q), thus we want to pad so that after modulation the resulting packet is a multiple of 128 samples.

@param ptk_byte_len: len in bytes of packet, not including padding.
@param samples_per_symbol: samples per bit (1 bit / symbol with GMSK)
@type samples_per_symbol: int

Packet Format ---- Access Code : Length : Payload : CRC
Real time scheduling support

```
gr_enable_realtime_scheduling()
{
    int policy = SCHED_FIFO;
    int pri = (sched_get_priority_max (policy) - sched_get_priority_min
               (policy)) / 2;
    int pid = 0;  // this process

    struct sched_param param;
    memset(&param, 0, sizeof(param));
    param.sched_priority = pri;
    int result = sched_setscheduler(pid, policy, &param);
    if (result != 0){
        if (errno == EPERM)
            return RT_NO_PRIVS;
        else {
            perror ("sched_setscheduler: failed to set real time priority");
            return RT_OTHER_ERROR;
        }
    }
    //printf("SCHED_FIFO enabled with priority = %d\n", pri);
    return RT_OK;
}
```

Tunnel.py:

```
........
r = gr.enable_realtime_scheduling()
    if r == gr.RT_OK:
        realtime = True
    else:
        realtime = False
        print "Note: failed to enable realtime scheduling"
........
```
MAC Implementation Framework

- Use Qualnet MAC code
  - Compile qualnet functions with *.c and *.h files in the gnu radio Makefile
  - Link the *.o files to the gnu radio shared libraries using ‘libtool’

- Wrapper function to handle interface between Qualnet and Gnu radio

- Allow for future protocols to just plug into gnu radio using the framework -
  -- Bridge gap between simulation and emulation environments
Tunnel.py Command line options

- **Basic**
  - Modulation scheme (GMSK, DQPSK, DBPSK),
  - Frequency of transmission (Depending on the daughter board),
  - Bitrate,
  - Tx and Rx subdevice (Depending on the side of the USRP),
  - Tx amplitude etc….

- **Expert**
  - Carrier Threshold,
  - TUN device to use,
  - Tx and Rx frequency,
  - Interpolation rate,
  - Samples per symbol,
  - FPGA decimation rate,
  - BW Time product,
  - Logging flow graph data, etc….
Matlab interface
Logging option

```python
# /////////////////////////////////////////////////////////////////////////////
#                              GMSK modulator
# /////////////////////////////////////////////////////////////////////////////
def _setup_logging(self):
    print "Modulation logging turned on."
    self._fg.connect(self.nrz,
                     gr.file_sink(gr.sizeof_float, "nrz.dat"))
    self._fg.connect(self.gaussian_filter,
                     gr.file_sink(gr.sizeof_float, "gaussian_filter.dat"))
    self._fg.connect(self.fmmod,
                     gr.file_sink(gr.sizeof_gr_complex, "fmmod.dat"))

# /////////////////////////////////////////////////////////////////////////////
#                            GMSK demodulator
# /////////////////////////////////////////////////////////////////////////////
def _setup_logging(self):
    print "Demodulation logging turned on."
    self._fg.connect(self.fmdemod,
                     gr.file_sink(gr.sizeof_float, "fmdemod.dat"))
    self._fg.connect(self.clock_recovery,
                     gr.file_sink(gr.sizeof_float, "clock_recovery.dat"))
    self._fg.connect(self.slicer,
                     gr.file_sink(gr.sizeof_char, "slicer.dat"))
```
Graph generation - GMSK
Graph generation - DQPSK
Current & Future work

Design & deploy a network testbed in GNU Radio to test MAC protocols like FLASHR

- Channel characteristics hard to simulate
- Overcomes assumptions made in simulations about the environment
GNURadio Applications

Presently implemented
- FM transmitter/receiver
- HDTV transmitter/receiver
- Spectrum Analyzer
- Oscilloscope
- Concurrent multi-channel receiver
- Advanced Television Systems Committee ATSC digital video standard
- Radio Astronomy Application based on USRP

In progress
- TiVo equivalent for radio
- Passive radar system
- Software GPS
- Distributed Sensor Networks
- RFID detector/reader
- Radio Direction finding routines
  etc...
Questions ?
GNU Radio links

- GNURadio Wiki
  - http://gnuradio.org/trac/wiki

- Installation Guides
  - http://www.kd7lmo.net/ground_gnuradio_install.html

- Online Tutorial
  - http://sdr.nd.edu/docs/
Thank You!