Spectrum Wars Documentation

Release 0.0.6

Tomaž Šolc

May 17, 2016

Contents

SpectrumWars is a work-in-progress programming game where players compete for bandwidth on a limited piece of a radio spectrum.

This is part of the effort in the FP7 CREW project to make a computer controlled version of the game that was demonstrated at various conferences (see for example FOSDEM 2014, Net Futures 2015). The aim is to replace a joystick-wielding human with a short, fun to write Python script.

This Python implementation for VESNA sensor nodes was written by Tomaž Šolc (tomaz.solc@ijs.si)

This work has been partially funded by the European Community through the 7th Framework Programme project CREW (FP7-ICT-2009-258301).

Contents:

CHAPTER 1

Introduction

SpectrumWars is a programming game where players compete for bandwidth on a limited piece of a radio spectrum. Its aim is to show the problems in spectrum sharing in an entertaining way. It captures the competition between various groups of users. Such competition is increasingly a factor in wireless communications as users demand more data in an even increasing variety of situations.

SpectrumWars extends the concept developed by P. Sutton and L. Doyle in The gamification of Dynamic Spectrum Access & cognitive radio by changing the role of the human competitors. In the initial concept, competitors are directly controlling transceivers via joystics. In this SpectrumWars implementation, they instead develop an algorithm that controls the transceivers in their place.

The competitive aspect of SpectrumWars was inspired in part by the DARPA Spectrum Challenge. However the goal here is to make the game as accessible and as fun as possible. The trade-off between realism for simplicity is heavily skewed towards the latter. For example, the interface between the player's code and the transceiver is greatly simplified, limiting the number of transceiver settings to just three: radio channel, bit rate and transmission power. This kind of a simplified toy-like interface was inspired by existing programming games, like the venerable *RobotWar* and its clones.

One aspect of real-life radio communications was not sacrified though: SpectrumWars games run on real hardware and use real radiofrequency spectrum. While a simulator is available to ease the development and debugging of player code, the SpectrumWars challenge runs on hardware provided by partners of the CREW project and takes place on real wireless testbeds.

1.1 Overview of the game

Competitors develop their algorithms using the Python scripting language. In a single game, two or more algorithms (*players*) compete with each other to transfer some useful data from a source to a destination as quickly and as reliably as possible. A good player for example will avoid interference from other players and the environment.

Players are aided in this task with the help of a spectrum sensor. In the game, the spectrum sensor is a centralized, simplified spectrum analyzer that is always available to the players. An algorithm can query it to get an up-to-date picture of the occupancy of the spectrum in the form of a power spectral density function.

The nature of the data being transferred is not directly known to players - it could conceivably be a machine-to-machine link sending sensor data, or it could be someone on a coffee break browsing their favorite social networking website. The player code only controls the connection and in fact does not need to concern itself with the payload part of the packets it is sending over the air.

Each player is given control of two transceivers (radio front-ends). For the purpose of the game, payload only needs to go from one transceiver (called *source*) to the other one (called *destination*). The players make use of a simple interface that provides basic control over the radio.

The separation between the source and the destination poses another challenge the players must overcome. There is no reliable back channel to use for synchronization between the two ends of the radio link. A rendezvous strategy is therefore required for all but the most simple algorithms.

Players are ranked by different statistics, like average packet loss and throughput. Different challenges are possible within the basic SpectrumWars framework. Some challenges might give more weight to the power efficiency of the players, while others might favor resilience against interference. Some might encourage players to intentionally interfere with competitors. Again some others might introduce an interfering spectrum user to the testbed where the game is played, but that is external to the game itself.

CHAPTER 2

Player's Guide

This part of the documentation covers topics interesting to readers wanting to implement player code. It describes the SpectrumWars player API and how to run and debug player code using a simulated testbed that does not require any special hardware connected to the computer.

2.1 Installing the simulator

The `pip install` command bellow will automatically install all missing Python packages. However, installing some larger dependencies from scratch can be a time consuming and error-prone process. It is therefore recommended that you install some of them using your Linux distribution package manager. Specifically:

- *numpy* (run `sudo apt-get install python-numpy` on Debian-based systems)
- *matplotlib* (run `sudo apt-get install python-matplotlib` on Debian-based systems)

To install the game controller, run:

```
$ pip install spectrumwars --user
```

To check if the installation was successful, try running the game controller:

```
$ spectrumwars_runner --help
```

If you get `command not found` error, check whether the scripts installed by `pip` are in your *PATH*. They are usually installed into `$HOME/.local/bin`.

2.2 Running player code

To run a game with a single player that is specified by source code in `examples/better_cognitive.py`:

```
$ spectrumwars_runner -l example.log examples/better_cognitive.py
```

You can add more players to the game by specifying more Python files to the command line.

Note: By default `spectrumwars_runner` uses a simulated testbed that does not require any special hardware. If you have testbed-specific hardware installed, you can specify the testbed to use using the `-t` command line argument. For example, to run the game using the VESNA testbed, use:

```
$ spectrumwars_runner -t vesna -l example.log examples/better_cognitive.py
```

In this case, `spectrumwars_runner` automatically finds any USB-connected VESNA nodes and assigns them randomly to players.

However, this is mostly intended for testbed developers. Playing SpectrumWars on a real testbed is usually done through a web interface.

While the game is running, you will see some debugging information on the console. In the end, some game statistics are printed out:

```
Results:
Player 1:
    crashed              : False
    transmitted packets  : 93
    received packets     : 51 (45% packet loss)
    transferred payload  : 12801 bytes (avg 981.4 bytes/s)

Game time: 13.0 seconds
```

If player code raised an unhandled exception at some point you will also see a backtrace. This should assist you in debugging the problem.

`spectrumwars_runner` allows you to set the game end conditions using the command-line arguments. Run `spectrumwars_runner --help` to see a list of supported arguments with descriptions. Also note that the capabilities of the `simulation` testbed can be customized to more closely resemble one of the real SpectrumWars testbeds. See *Simulated testbed*.

In addition to the ASCII log that is printed on the console, the game controller also saves a binary log file to `example.log`. The binary log contains useful debugging information about events that occurred during the game. You can visualize the log by running:

```
$ spectrumwars_plot -o example.out example.log
```

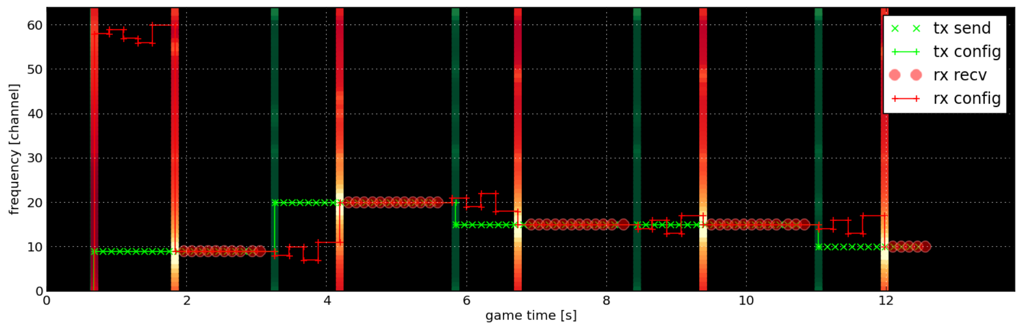
This creates a directory `example.out` with a few images in it. One visualization is created for each player participating in the game. These are named `player0.png`, `player1.png` and so on, using the same order as it was used on the `spectrumwars_runner` command line. One additional visualization named `game.png` is created showing the overall progress of the game.

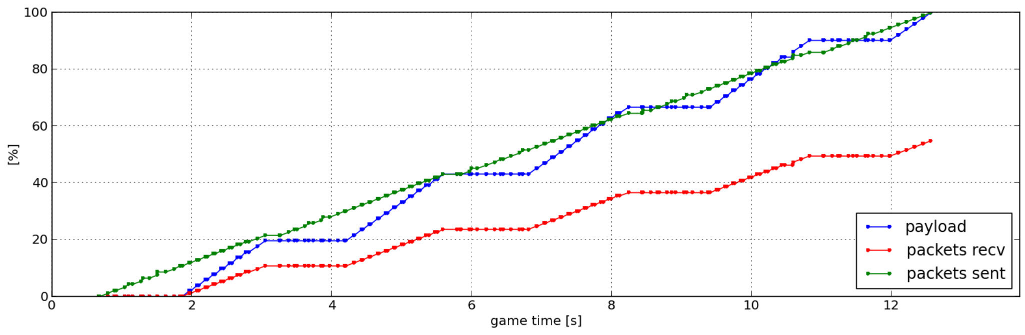
See Understanding the visualizations on how to read the graphs produced by *spectrumwars_plot*.

2.3 Understanding the visualizations

2.3.1 Player's visualization

One image per player is produced by `spectrumwars_plot` in the specified output directory. Images are saved to files named `player0.png`, `player1.png`, etc.





The upper graph with the black background shows the progress of the game in a time-frequency diagram. Game time is on the horizontal axis and frequency channels are on the vertical. Key events in the game are displayed in this diagram with the focus on the current player.

- Red color marks events related to this player's `Destination` class,
- green color marks events related to this player's `Source` class and
- gray color marks events related to other player's transceivers.

Since only one player participated in this game, there are no gray color markers on the diagram shown above. The behavior of the single player can be seen from the following markers:

- Green crosses show transmitted packets from the player's `Source` class. These correspond to calls to the `send()` method.
- Red circles show packets, that were successfully received by the player's `Destination` class.
- Thick green and red vertical lines show spectral scans by the source and destination respectively. These correspond to calls to the `get_status()` method, or when the `status_update()` event happens. The lines vary slightly in color to show the result of the spectral scan - lighter color means a higher detected signal level on the corresponding channel.
- The small crosses connected with a thin green and vertical lines show the currently tuned frequency of the source and destination respectively. The lines shift in frequency for each call to the `set_configuration()` method.

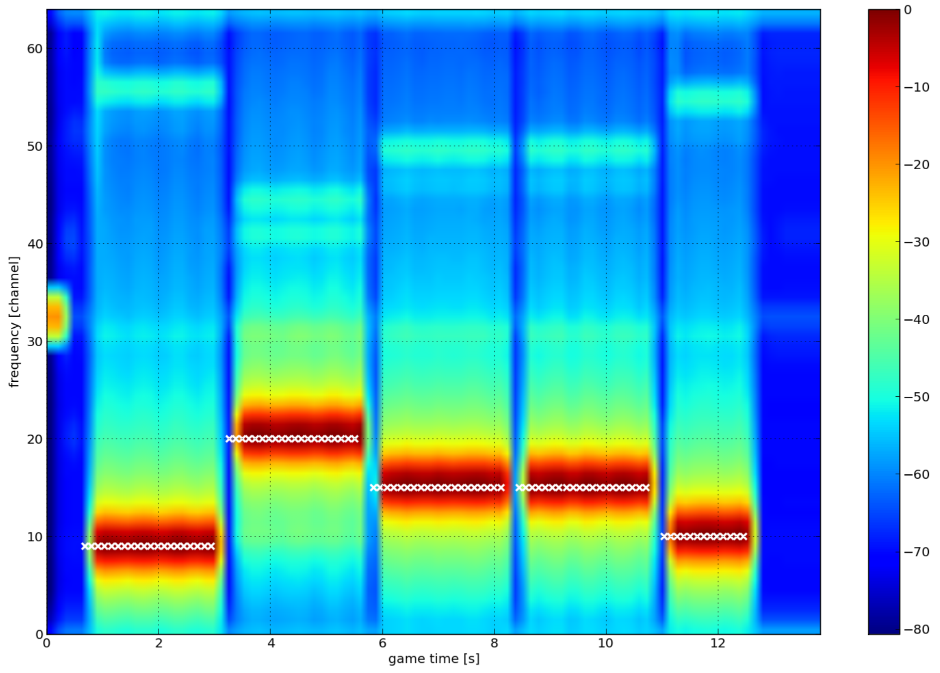
Note: Only packet transmissions are shown for other players.

Reading the specific diagram above, you can see that the source first started transmitting near channel 10. After around 3 seconds, it performed a spectral scan and shifted the frequency to channel 20. The destination on the other hand, attempted first to unsuccessfully receive packets around channel 60. Then it performed a spectral scan at around 2 second mark. After the scan it tuned to the source's channel and started to successfully receive packets. This continued until the source jumped to channel 20, after which the destination started changing channels again in an attempt to restore packet reception.

The bottom graph shows progress of some performance indicators: percentage of transferred payload, transmitted and received packets. These are relative to the total payload and packet counts in the game.

2.3.2 Game visualization

One image per game is created by `spectrumwars_plot` in the specified output directory. Image is saved to a file named `game.png`.



Similar to the upper graph in the per-player visualization, this graph shows a time-frequency diagram. The color on the diagram shows signal level, as reported by the actual spectrum sensor, for each channel and moment in time while the game was running.

The color bar on the right shows the mapping between the color and the specific value that would be seen by player code at that time and channel if it called the `get_status()` method.

Exact time and frequency of packet transmissions of all players in the game are shown superimposed over the diagram using small white crosses.

2.4 Scoring

At the moment, SpectrumWars doesn't use a single scoring function. Players may be ranked by different criteria, depending on a specific competition. Game controller currently records the following statistics for each player in a game:

Packet loss Packet loss is defined as:

```
   _     Ntx - Nrx
   PL = ---------
            Ntx
```

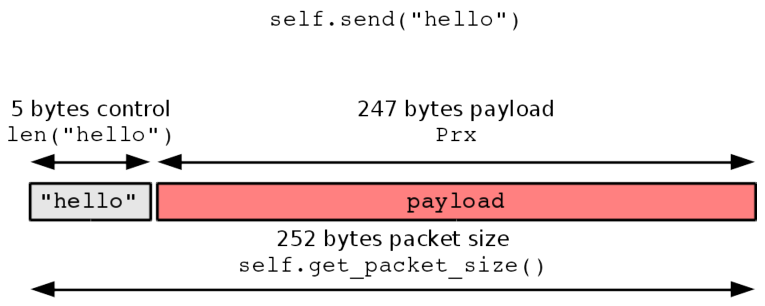
Where Ntx is the total number of packets transmitted by the player's source during the game and Nrx is the total number of packets received by the player's destination during the game.

Note that packets with control data in the direction from the *destination* to the *source* do not count.

Throughput Throughput is defined as:

```
   _      Prx     bytes
   TP = ----- [ ----- ]
           T        s
```

Where Prx is the total amount of payload data received by the player's destination in bytes and T is the duration of the game in seconds.



2.5 Implementing a transceiver

The Python source code file you provide to SpectrumWars (called a *player*) should define two subclasses of the base Transceiver class: one for the source (named Source) and one for the destination (named Destination). Game controller makes one instance of the source class and one instance of the destination class.

From the standpoint of the programming interface, the source and destination classes are identical (e.g. destination can also send data to the source). However in the game, their role differs: payload data is only sent from the source to the destination. This means that statistics like packet loss and throughput which are used in ranking the players are only counted in that direction.

The Transceiver class interface has been designed to accomodate two programming styles: procedural programming and event-based programming.

This is how a simple procedural destination looks like:

```python
class Destination(Transceiver):
   def start(self):
      # do some setup
      self.set_configuration(...)

      # loop until the game ends
      while True:
         # ask for the most recent spectrum scan and game status
         status = self.get_status()
         ...

         for packet in self.recv_loop():
            # do something with queued-up packet data
            ...
```

And this is how an identical event-based destination looks like:

```
class Destination(Transceiver):
   def start(self):
      # do some setup
      self.set_configuration(...)

   def recv(self, packet):
      # do something with received packet data
      ...

   def status_update(self, status):
      # do something with the updated spectrum scan
      ...
```

Both styles are compatible and you can use a combination of both if you wish. If you are unsure which one to use, we recommend the procedural style.

2.5.1 Class reference

class **`Transceiver`**

You should override the following methods in the `Transceiver` class to create your source and destination classes. Do not override or use any members prefixed with an underscore (_). These are for internal use only.

`start``()`

Called by the game controller upon the start of the game. This method can perform any set-up required by the transceiver.

Once this method returns, the game controller's event loop will start calling `recv()` and `status_update()` methods as corresponding events occur. You can however prevent this method from returning and use it to implement your own loop, as in the procedural example above. Of course, other players will not wait until your `start()` returns.

If left unimplemented, this method does nothing.

`recv``(`*packet*`)`

Called by the game controller when the transceiver receives a packet (e.g. called on the receiver side when the transmitter side issued a send() and the packet was successfully received).

Note that you do not need to override this method for the received payload to be counted towards your score. Overriding is only useful when you want to respond to a successfull reception of a packet or you want to do something with the data in the packet sent by the transmitter.

`packet` is a `RadioPacket` object containing the string that was passed to `send()` on the transmitting side.

If left unimplemented, this method does nothing.

`status_update``(`*status*`)`

Called by the game controller periodically with updates on the state of the game.

`status` is a `GameStatus` object.

If left unimplemented, this method does nothing.

From these overriden methods you can call the following methods to control your transceiver:

set_configuration(*frequency*, *bandwidth*, *power*)

Set up the transceiver for transmission or reception of packets on the specified central frequency, power and bandwidth.

`frequency` is specified as channel number from 0 to N-1, where N is the value returned by the `get_frequency_range()` method. Central frequencies of channels are hardware dependent.

`bandwidth` is specified as an integer specifying the radio bitrate and channel bandwidth in the interval from 0 to N-1, where N is the value returned by the `get_bandwidth_range()` method. Exact interpretation of this value is hardware dependent. Higher values always mean higher bitrates and wider channel bandwidths.

`power` is specified as an integer specifying the transmission power in the interval from 0 to N-1, where N is the value returned by the `get_power_range()` method. Exact interpretation of this value is hardware dependent. Higher values always mean **lower** power.

See Testbed reference for interpretations of these values.

Invalid values will raise a `RadioError` exception.

Note: While specific meanings of these settings are hardware specific, you can assume that your receiver and transmitter will only be able to communicate successfully if both use the same `frequency` and `bandwidth` settings. The `power` setting is less critical, but higher power will usually lead to more reliable communication.

get_configuration()

Returns a `(frequency, bandwidth, power)` tuple containing the current transmission or reception configuration.

send(*data=None*)

Send a data packet over the air. On the reception side, the `recv()` method will be called upon the reception of the packet.

`data` is an optional parameter that allows inclusion of an arbitrary string into the packet. On the reception side, this string is passed to the `recv()` method in the `data` field of the `RadioPacket` object.

Note that the length is limited by the maximum packet size supported by the radio (as returned by `get_packet_size()`). Longer strings will raise a `RadioError` exception.

Upon successfull reception of the packet on the receiver side, `n` bytes are counted towards the player's score, where `n = packet_size - len(data)`.

get_status()

Returns the current state of the game in a `GameStatus` object.

recv_loop(*timeout=1.*)

Returns an iterator over the packets in the receive queue. Packets are returned as `RadioPacket` objects.

`timeout` specifies the receive timeout. Iteration will stop if the queue is empty and no packets have been received for the specified number of seconds (note that floating point values < 1 are supported)

The following methods can be used to query the capabilities of the testbed. You can use them if your want to automatically adapt your algorithm to the testbed it is running on. If you are targeting just one testbed, you can ignore this part and look at the Testbed reference.

get_frequency_range()

Returns the number of available frequency channels.

get_bandwidth_range()

Returns the number of available bandwidth settings.

get_power_range()
Returns the number of available transmission power settings.

get_packet_size()
Returns maximum number of bytes that can be passed to `send()`.

class **GameStatus**
The `GameStatus` class contains the current status of the game. The following attributes are defined:

spectrum
This attribute contains the current state of the radio spectrum.

`spectrum` is a list of floating point values. Each value is received power in a frequency channel in decibels, as seen at the antenna of the spectrum sensor observing the game. Frequency channels are the same as ones used by `set_configuration()`. Length of the list is equal to the value returned by `get_frequency_range()`. Reported power levels are relative.

For example, if `spectrum[10] == -60`, that means that -60 dB of power have been seen by the sensor on the radio channel obtained by `set_configuration(10, x, y)`.

Note: `send()` radio transmissions typically occupy several radio channels around the specified central frequency specified by `set_configuration()`. Number of occupied channels depends on the specified bitrate.

class **RadioPacket**
A `RadioPacket` object is passed to the receiving transceiver for each successfully received packet. The following attributes are defined:

data
This attribute contains the string that was passed to the `send()` method on the transmitting side.

2.6 An annotated example

The `examples` directory in the SpectrumWars source tree contains a number of player code examples (see it on GitHub). You are encouraged to explore the example code and study how it works. To help you, `better_cognitive.py`, one of the examples, is explained step-by-step in this section.

Refer to the *Class reference* for details on SpectrumWars-specific method calls used. The examples also use *numpy* to simplify some parts of the code. See NumPy reference for details.

```
# Import the Transceiver base class.
from spectrumwars import Transceiver

# Also import NumPy to get some convenient array functions.
import numpy as np

# Two modules from the Python standard library we'll use.
import random
import time

# First, let's write the code that controls our source.
class Source(Transceiver):

    # We use the procedural style in this example, so we only override the
    # start() method. This way our code gets called right at the start of
    # the game. Our source will not leave this method until the game
    # ends.
    def start(self):

        # We simply make an infinite loop. We don't need to care what happens
        # when the game ends - game controller takes care of cleaning up
        # after us.
        while True:

            # Ignore this delay for now. Real testbeds have various delays
            # when sending packets or when sensing the spectrum. Sometimes it
            # helps to artificially slow down your algorithm.
            time.sleep(.2)

            # self.get_status().spectrum returns a list of received signal
            # strength indicators (RSSI) for all channels in the testbed.
            # Index into this array directly translates to the radio channel
            # number.
            #
            # We convert the result to a NumPy array for convenience.
            spectrum = np.array( self.get_status().spectrum )

            # These two lines take the RSSI list and select one channel at
            # random from 20 channels that have the lowest signal strength.
            chl = np.argsort(spectrum)
            ch = chl[random.randint(0, 20)]

            # Now we tune the radio to the selected channel. We also select
            # the slowest (and most reliable) bitrate setting and the
            # strongest transmission power.
            self.set_configuration(ch, 0, 0)

            # Next, we transmit 20 packets on the selected channel. We don't
            # add any control data to the packets, so the complete packet is
            # filled with payload by the game controller.
            for n in xrange(20):
                self.send()

                # We delay a little bit the transmission of packets.
                time.sleep(.05)

            # After 20 packets have been transmitted, our loop rolls around
            # for another iteration. We again check the spectrum occupancy,
            # select one of the channels that appear to be least occupied and
            # transmit another 20 packets.
            #
            # The spectrum sensor has some averaging. If we would loop
            # immediately from self.send() to self.get_status(), the spectral
            # scan would still contain the trace of our own packets. Hence the
            # 200 ms delay at the start of the loop to let our packets fall
            # out of the averaging window and make it possible to select the
            # same channel again.


# Now for the destination side of the code.
class Destination(Transceiver):

    # Again, destination spends the duration of the game in the start() method.
    def start(self):
        # Since the first thing we do in the source is a 200 ms delay,
        # there is no point in trying to receive anything earlier than that.
        time.sleep(.2)

        # Another infinite loop.
        while True:

            # Wait a bit more to be sure that the source is transmitting
            # at this point and that its packets have been picket up by the
            # spectrum sensor.
            time.sleep(.1)

            # Use the same method as in the source to get a NumPy array
            # containing RSSI values for all channels.
            spectrum = np.array( self.get_status().spectrum )

            # This line uses a similar argsort trick as in the source.
            # We want an array of channel numbers, sorted with the channel with
            # the highest signal strength on top.
            chl = np.argsort(spectrum)[::-1]

            # For each channel of the top five by signal strength...
            for ch in chl[:5]:

                # ... tune the radio to that channel. Set bitrate to the same
                # one as used by the source.
                #
                # We also set the transmit power to the higher setting. However
                # we don't transmit anything from the destination side in this
                # example.
                self.set_configuration(ch, 0, 0)

                # On the selected channel, wait 200 ms for a packet.
                for packet in self.recv_loop(timeout=.2):

                    # We don't do anything with the received packet - the
                    # source did not include any control data that would be
                    # interesting to us.
                    #
                    # Game controller takes care of the payload data automatically.
                    #
                    # If a packet has been received within 200 ms, the inner for
                    # loop rolls around and waits 200 ms for another packet.
                    pass

                # If a packet has not been received for 200 ms, the outer for
                # loop tries with the next most occupied channel.

            # If reception has been unsuccessful, the while loop rolls around
            # and performs another spectral scan, repeating the process.
```

At this point, you should try running this example in the simulation a few times and check the resulting time-frequency diagrams. Try to run it in a game competing with some other example players. Find its flaws and see how it can be improved.

2.7 Testbed reference

2.7.1 VESNA

VESNA testbed uses VESNA sensor nodes with narrow-band radios as transceivers in the 2.4 GHz band. An Ettus Research USRP N200 is used as a spectrum sensor.

Use `-t vesna` to with `spectrumwars_runner` to use this testbed.

parameter	value
Maximum packet length	252 bytes
Number of frequency channels	64
Number of bandwidth settings	4 (see *Interpretation of bandwidth settings*)
Number of transmission power settings	17 (see *Interpretation of transmission power settings*)

Central frequency of a channel can be calculated using the following formula:

```
f = 2400.0 MHz + <chan> * 0.1 MHz
```

Table 2.1: Interpretation of bandwidth settings

`bandwidth`	bitrate
0	50 kbps
1	100 kbps
2	200 kbps
3	400 kbps

Table 2.2: Interpretation of transmission power settings

`power`	dBm
0	0
1	-2
2	-4
3	-6
4	-8
5	-10
6	-12
7	-14
8	-16
9	-18
10	-20
11	-22
12	-24
13	-26
14	-28
15	-30
16	< -55

2.7.2 Simulated testbed

Simulated testbed uses a software simulation to run the game. No special hardware is required. This is useful when developing player code.

This testbed is used by default by `spectrumwars_runner`, if no `-t` argument is specified.

Capabilities of this testbed can be customized using the following keyword arguments (use `-Okeyword=value` in the `spectrumwars_runner` command-line to modify their values from default):

keyword	meaning	default	unit
packet_size	Maximum packet length	1024	bytes
frequency_range	Number of frequency channels	64	
bandwidth_range	Number of bandwidth settings	10	
power_range	Number of transmission power settings	10	
send_delay	Time for sending one packet	0.100	s

Note that the simulation of the radio environment is greatly simplified:

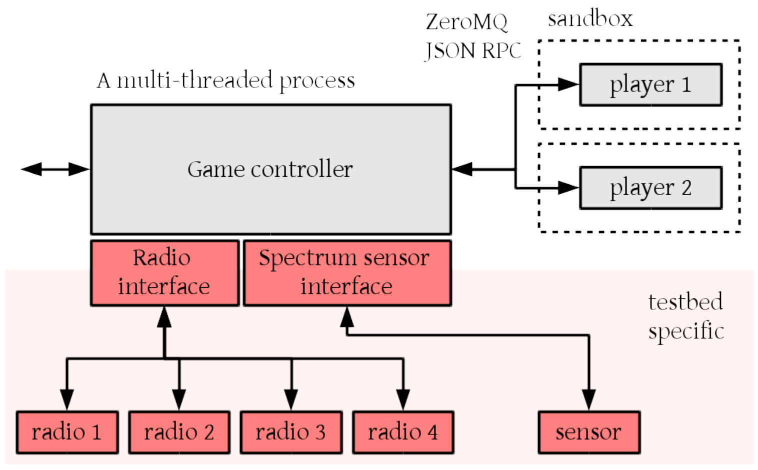
- A packet occupies only the channel it is sent on.
- Sending of all packets takes the same amount of time (`send_delay`), regardless of `bandwidth` setting.
- Only very simple collision detection is implemented. If transmission of two packets commences within the `send_delay` of each other, the first packet will be successfully delivered to its recepient, while the second will be discarded.
- Spectrum sensing shows higher received power on channels with recent packet transmissions.
- Transmission power setting is ignored.

For the impatient, the fast track to get started is:

- Check An annotated example and examples on GitHub and
- experiment with the simulator as described in Installing the simulator and Running player code.

CHAPTER 3

Developer's Guide



This part of the documentation covers parts that are interesting for testbed operators wanting to add support for their hardware to SpectrumWars, administrators wanting to deploy SpectrumWars on their testbed and SpectrumWars developers.

3.1 Installation instructions

3.1.1 Getting the source

Up-to-date development version is available on GitHub at:

https://github.com/avian2/spectrumwars

You can download the source using the following command:

```
$ git clone https://github.com/avian2/spectrumwars.git
```

If you intend to do development, it's best if you make your own fork of the repository on GitHub.

SpectrumWars releases are also available from PyPi.

3.1.2 Setting up the testbed

If you would like to run Spectrum Wars on real hardware, you first have to setup the testbed. Follow the instructions in the appropriate section of Testbed-specific installation instructions.

Skip this step if you would only like to use Spectrum Wars with a simulated testbed.

3.1.3 Installing game controller

You need the following packages installed:

- jsonrpc2-zeromq (`pip install jsonrpc2-zeromq --user`)
- numpy (`apt-get install python-numpy`)
- matplotlib (`apt-get install python-matplotlib`)

To install, run:

```
$ cd controller
$ python setup.py install --user
```

To run unit tests shipped with the code:

```
$ python setup.py test
```

Note that to run the testbed-specific tests, you need to have the testbed hardware connected and working at this point.

Tests that require hardware or optional external dependencies that were not found on the system are skipped (check the console output for any lines that say `skip`).

Note: If you get errors like `SandboxError: Can't find 'spectrumwars_sandbox' in PATH`, check whether the scripts installed by `setup.py` are in your *PATH*. They are usually installed into `$HOME/.local/bin` if you used the `--user` flag as suggested above.

3.1.4 Building HTML documentation

You need the following software installed to build documentation:

- Sphinx (`apt-get install python-sphinx`)

To rebuild documentation run:

```
$ cd docs
$ make html
```

Index page is created at `_build/html/index.html`.

3.2 Testbed-specific installation instructions

3.2.1 VESNA

VESNA testbed uses VESNA sensor nodes designed by the Institute Jožef Stefan.

Required hardware

- One USRP N200 connected over a gigabit Ethernet interface to be used as a spectrum sensor. Use default network settings (use 192.168.10.1 for the computer's IP)

  Current setup uses SBX daughterboard, a 2.4 GHz antenna.

- VESNA sensor nodes, connected through a powered USB hub. As many as you need (two nodes per player).

  Nodes should consist of a SNC core board and a SNE-ISMTV-2400 radio board.

  If you need to upload firmware, you will also need a SNE-PROTO board and a Olimex ARM-USB-OCD programmer. For debugging, a serial-to-USB converter connected to VESNA's USART1 is recommended.

Note: As of 0.0.3, `spectrumwars_runner` no longer uses VESNA nodes connected over serial-to-USB converters.

Firmware compilation

Firmware has to be uploaded to VESNA sensor nodes before they can be used with Spectrum Wars. Firmware source code is stored in the `firmware/` subdirectory.

You will need the ARM toolchain installed. See https://sensorlab.github.io/vesna-manual/ for instructions. These steps assume you are using Linux and that the command line tools are properly set up.

You will also need a checkout of the `vesna-drivers` repository. Current packet driver in the `master` branch is very unstable and unsuitable to be used with this firmware. It is recommended that you use the `spectrumwars` branch from the following repository:

https://github.com/avian2/vesna-drivers

First make sure that the `VESNALIB_LOCATION` in the `Makefile` points to the directory containing the `vesna-drivers` git repository:

```
$ cd firmware
$ grep VESNALIB_LOCATION= Makefile
```

To compile the firmware run:

```
$ make
```

To upload the firmware, make sure you have the Olimex ARM-USB-OCD connected to the node and run:

```
$ make install
```

To test the firmware, connect two nodes using the mini-USB connector and run:

```
$ cd ../controller
$ python setup.py test -s tests.test_radio
```

Note: In case of problems, there are some debugging options available on top of `vsndriversconf.h`. See also Firmware interface.

Additional dependencies for game controller

In addition to packages listed in Installation instructions, the following additional packages are required to use Spectrum Wars game controller with the VESNA testbed:

- GNU Radio with UHD (GNU Radio version 3.7.5.1 is known to work) - http://gnuradio.org
- gr-specest - https://github.com/avian2/gr-specest
- pyudev (`apt-get install python-pyudev`)
- pyserial (`apt-get install python-serial` or `pip install pyserial --user`)

Also, make sure that `/dev/ttyACMxx` devices are accessible to the user running the game controller. Typically, this requires adding the user to the `dialout` group. For example:

```
# adduser myuser dialout
```

3.3 Firmware interface

In VESNA implementation of the SpectrumWars game, one or more sensor nodes are connected to the game controller (e.g. a Linux running computer) over the USB.

USB carries a simulated serial line. Normally, the SpectrumWars firmware uses VESNA's mini-USB interface to expose a standard USB CDC endpoint identified by product string "VESNA SpectrumWars radio" (Linux typically associates a device file named `/dev/ttyACMx` with these endpoints).

Alternatively, a setting in `vsndriversconf.h` allows compilation of firmware that uses RS-232 serial line instead of USB CDC. In this case, the firmware's interface is exposed on VESNA's USART1 connector (115200 baud, 8 data bits, 1 stop bit, no parity). A serial-to-USB converter can be used to connect such a node to the game controller. Serial-to-USB converters are usually associated with `/dev/ttyUSBx` device files on Linux.

The testbed controller controls the sensor node using a terse, ASCII base protocol. Commands are kept short to keep the protocol reasonably fast even when used over a relatively slow serial line. ASCII has been chosen over a binary protocol to aid in debugging (i.e. the node can be controlled manually using a standard serial terminal for development and debugging purposes)

Protocol consists of atomic messages: commands (sent from the controller to the node) and responses (sent from the node to the controller). Each message starts with a unique ASCII alphabet character denoting the type of the message, has optional space separated parameters and ends with ASCII line feed character ("\n").

Following commands can be sent from the controller to the node:

a *<address>* Set the radio address. <*address*> is a hexadecimal integer in the range 0-255 containing the MAC address. Node will only receive packets addressed at the configured MAC address and will silently ignore others.

c *<chan> <bw> <power>* Set the radio channel, bandwidth and power. Parameters are hexadecimal integers with the following meanigs:

<*chan*> radio frequency channel to tune to. Valid channels are from 0 to 255. Central frequency is calculated using the following formula:

```
    f = 2400.0 MHz + <chan> * 0.1 MHz
```

<*bw*> channel bandwidth setting to use, starting with 0. This setting affects the channel filter and modem bitrate according to the following table:

<bw>	bitrate [kbps]	channel filter [kHz]
0	50	100
1	100	200
2	200	400
3	400	800

<*power*> power amplifier setting to use, starting with 0. This setting affects the transmission power according to the following table:

<power>	transmit power [dBm]
0	0
1	-2
2	-4
3	-6
4	-8
5	-10
6	-12
7	-14
8	-16
9	-18
10	-20
11	-22
12	-24
13	-26
14	-28
15	-30
16	< -55

Regardless of the settings, minimum-shift keying modulation is used.

t *<address>* *<data>* Transmit a packet using the previously set radio parameters.

<*address*> is a hexadecimal integer in the range 0-255 containing the MAC address of the recipient.

<*data*> is a hexadecimal string containing the data to be transmitted in the packet. Two hexadecimal digits per byte. The length of the string can be between 1 and 252 bytes.

For example, sending packet with ASCII content "hello" to address 1:

```
t 01 68656c6c6f
```

Node can respond with the following responses:

O Last command was successfully executed.

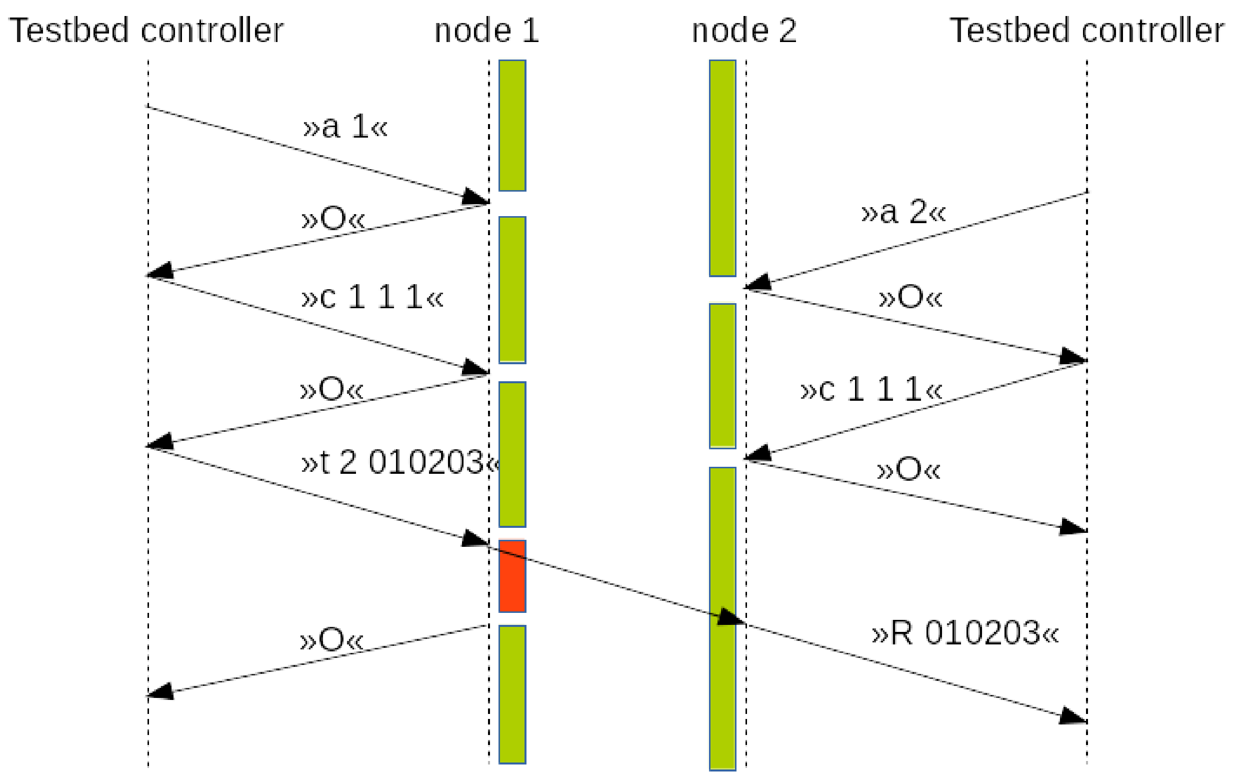
E *<message>* Last command resulted in error. <*message*> is an ASCII string describing the error.

R *<data>* Node received a packet. <*data*> is a hexadecimal string containing the data in the packet.

Radio does CRC checking in hardware and silently drops corrupted packets. Hence it is very likely that the <*data*> string is identical to the one passed to the corresponding *t* command.

Any messages not conforming to this response format should be ignored by the controller. In practice, nodes can emit additional debugging information over this channel (see settings in `vsndriversconf.h`)

By default, the node's radio is kept in receive mode. Receive mode is temporarily turned off during reconfiguration. After receiving a transmit command, the node switches the radio to transmit mode, transmits the single packet and switches back to receive mode.



🟩 radio in RX mode

🟥 radio in TX mode

3.4 Adding support for a new testbed

To add support for a new testbed, the following tasks need to be done:

Add testbed specific code blocks Add a module with the name like `spectrumwars.testbed.xxx`. This module should define two classes:

- `Testbed`, a subclass of `TestbedBase`, and
- `Radio`, a subclass of `RadioBase`.

Add testbed specific unit tests Add tests to a Python file with the name like `tests/test_xxx.py`. Please make the tests automatically skip themselves if the testbed-specific hardware is not connected (e.g. raise the `unittest.SkipTest` exception)

Add testbed documentation Add testbed documentation for players to `docs/reference.rst`. Add any testbed-specific installation instructions to `docs/installtestbed.rst`.

3.4.1 Class reference

class `Testbed`

`Testbed` objects represent a physical testbed that is used to run a game. Unless stated otherwise, subclasses should override all of the methods and attributes described below.

The class constructor can take optional string-typed keyword arguments. These can be specified in `spectrumwars_runner` using the `-O` arguments.

`RADIO_CLASS`
Should be set to the subclass of the `RadioBase` class that is used by the testbed (i.e. *`Radio`* in the testbed's module)

`get_radio_pair``()`
Returns a `rxradio, txradio` tuple. `rxradio` and `txradio` should be instances of *`RADIO_CLASS`*.

This method is called multiple times by the game controller, once for each player to obtain the interfaces to player's radios. It is called before the game starts, and before the call to *`start()`*.

`start``()`
Called once, immediately before the start of the game.

`stop``()`
Called after the game concluded. This method should perform any clean-up tasks required by the testbed (e.g. stopping any threads started by *`start()`*.

`get_frequency_range``()`
Returns the number of frequency channels available to player's code. The value returned should not change during the lifetime of the object.

Corresponds to *`Transceiver.get_frequency_range()`*.

`get_bandwidth_range``()`
Returns the number of bandwidth settings available to player's code. The value returned should not change during the lifetime of the object.

Corresponds to *`Transceiver.get_bandwidth_range()`*.

`get_power_range``()`
Returns the number of transmission power settings available to player's code. The value returned should not change during the lifetime of the object.

Corresponds to *`Transceiver.get_power_range()`*.

`get_spectrum``()`
Returns the current state of the radio spectrum as a list of floating point values.

The value returned by this method gets assigned to *`GameStatus.spectrum`*.

See also *`usrp_sensing.SpectrumSensor`*.

`time``()`
Returns the current testbed time in seconds since epoch as a floating point number. Selection of an epoch does not matter. Game controller requires only that time increases monotonically.

By default it returns `time.time()`, which should be sufficient for most testbeds.

class `Radio`
`Radio` objects represent a player's interface to a single transceiver. Unless stated otherwise, subclasses should override all of the methods described below.

`PACKET_SIZE`
Set to the maximum length of a string that can be passed to the `send()` method.

Approximately corresponds to *`Transceiver.get_packet_size()`*. Game controller adds a header to separate control data from payload which adds an overhead of a few bytes. Because of this, the player visible maximum packet size will be lower.

set_configuration(*frequency*, *bandwidth*, *power*)
Set up the transceiver for transmission or reception of packets on the specified central frequency, power and bandwidth.

`frequency` is specified as channel number from 0 to N-1, where N is the value returned by the *Testbed.get_frequency_range()* method.

`bandwidth` is specified as an integer specifying the radio bitrate and channel bandwidth in the interval from 0 to N-1, where N is the value returned by the *Testbed.get_bandwidth_range()* method. Higher values mean higher bitrates and wider channel bandwidths.

`power` is specified as an integer specifying the transmission power in the interval from 0 to N-1, where N is the value returned by the *Testbed.get_power_range()* method. Higher values mean **lower** power.

Corresponds to *Transceiver.set_configuration()*.

binsend(*bindata*)
Send a data packet over the air.

`bindata` is a binary string with the data to be included into the packet. Length of `bindata` can be up to *PACKET_SIZE*.

Corresponds to *Transceiver.send()*. Note that the game controller packs the packet with payload, so `bindata` will not be identical to the `data` string passed to *Transceiver.send()*.

binrecv(*timeout=None*)
Return a packet from the receive queue.

`timeout` specifies the receive timeout in seconds. If no packet is received within the timeout interval, the method raises `RadioTimeout` exception.

Upon successfull reception, the method should return a binary string. The returned string should be equal to the `bindata` parameter that was passed to the corresponding `send()` call.

Note: There is no way for the `Radio` class to push packets towards the game controller. Instead, the game controller polls the radio for received packets by calling `recv()` method, as instructed by player's code. Hence it is in most cases necessary that the actual packet reception happens in another thread (started typically from *Testbed.start()*) and that the received packets are held in a queue until the next `recv()` call.

Corresponds to *Transceiver.recv()*. Note that the game controller unpacks the payload from the packet before passing it to *Transceiver.recv()*.

class `usrp_sensing.`**SpectrumSensor**(*base_hz*, *step_hz*, *nchannels*, *time_window=200e-3*, *gain=10*)
`usrp_sensing.SpectrumSensor` is a simple, reusable spectrum sensor implementation using a USRP device.

The sensing algorithm is inspired by a real-time signal analyzer. The recorded samples are converted into power spectral density using continuous end-to-end FFTs with no blind time (and no overlap of the FFT windows). The spectral power density is then averaged over a time window.

The algorithm is very CPU intensive. Using a 2.7 GHz CPU, it will be able to sense at most 64 channels (even if USRP frontend bandwidth would allow for more).

Sensing in this way is necessary because the radios usually have a very low duty cycle (e.g. a "while True: send()" has only around 10% duty cycle on the VESNA testbed). If we would only take one sample the spectrum when players request it, it would mostly appear empty. Hence the need to take a moving average if sensing is to be useful for detecting player transmissions.

base_hz is the lower bound of the frequency band used in the game in hertz. *step_hz* is the width of each channel. *nchannels* is the number of channels used in the game. The values for these parameters should be chosen so that the channel frequencies correspond to the channels used by the testbed's `Radio` class:

```
-------------------------------> frequency (Hz)

+---+---+     +---+
| 0 | 1 | ... | n |  (channels used in the game)
+---+---+     +---+

|---| <- step_hz

|-----------------| <- step_hz * nchannels

^
|

base_hz
```

time_window defines the length of the moving average filter in seconds. The value depends on how often players can look up the current state of the spectrum. In most cases it should be longer than the period of `Transceiver.status_update()` events in the event-based model.

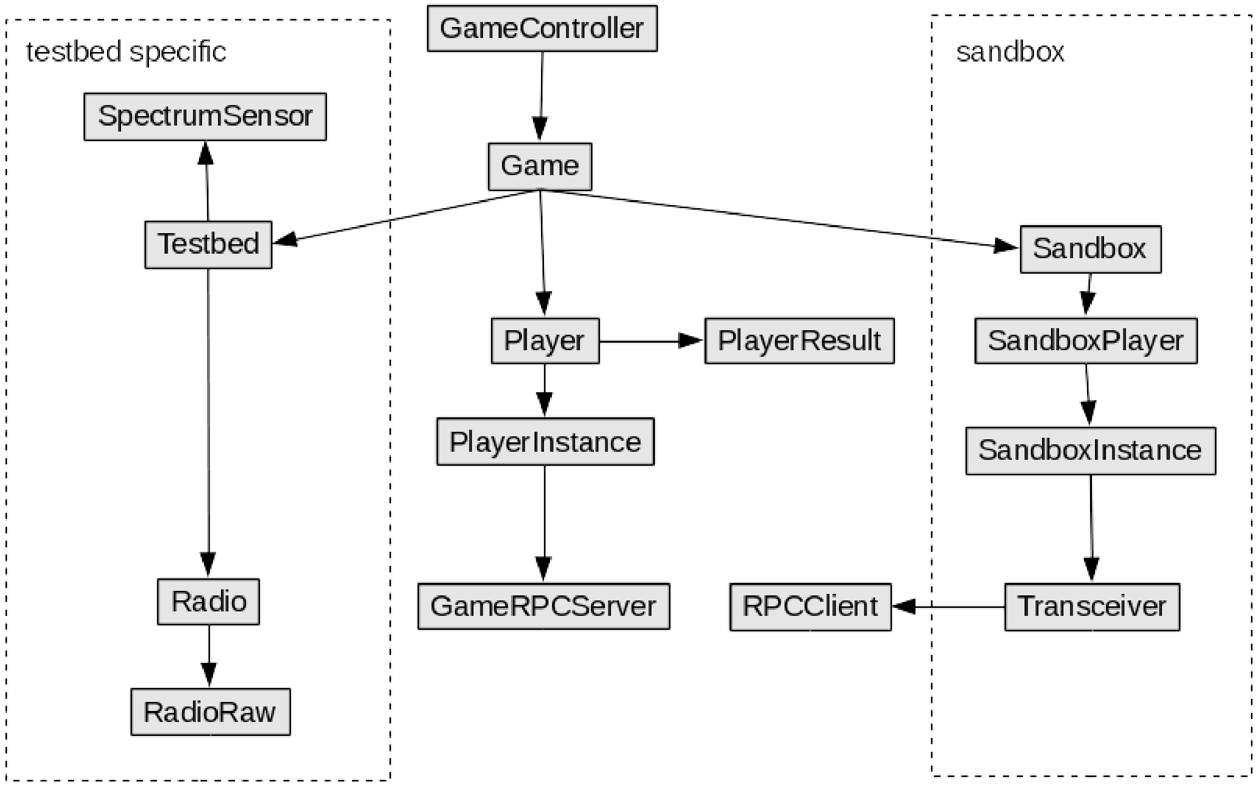
start()
Start the worker thread. Should be called before first call to `get_spectrum()`

stop()
Stop the worker thread.

get_spectrum()
Returns the current state of the radio spectrum as a list of floating point values. Length of the list is equal to *nchannels*.

The value returned by this method can be directly used as the return value of `Testbed.get_spectrum()`.

3.5 Implementation notes



- 64 channels is a very generous portion of the spectrum for this game. A single channel could in theory accomodate 10 players with very little interferrence.
- Currently, `spectrumwars_runner` runs player's code in a separate processes (provided by `spectrumwars_sandbox` executable). The processes communicate through ZeroMQ JSON RPC. This provides some isolation between players and the game controller. It prevents simple ways of cheating that would be possible if code would share the same Python interpreter. It also gracefully handles infinite loops and most accidental errors.

  This is not, however, robust against more sophisticated malicious code. There is currently nothing preventing one user from accessing the RPC interface inteded for another user. There is also nothing preventing player's code from accessing the network, filesystem or consuming excessive amounts of memory. These limitations must be implemented on the operating system level and current code makes no attempt to implement them.

  In the future, more sophisticated sandbox methods might be implemented (e.g. running player's code in a virtual host). Since all communication between player's code in `Transceiver` class and the game controller already occurs over RPC, this should not require further modifications to the game controller.
- Using sensor nodes connected directly over USB (instead of using serial-to-USB converters) greatly simplifies the setup - apart from a powered USB hub, there is no need for having a converter plus a separate power supply for each node.

  However, USB CDC implementation on VESNA is still prone to occasional data loss. This is visible as errors in communication between the game controller and the radio (e.g. `RadioTimeout` exceptions or truncated packet payload). At the moment this is rare enough for direct USB connection to be considered usable in practice.
- I believe that in the final user interface for this game, it is crucial that both console log of the running game and the visualized timeline are presented to each player. Without this kind of feedback it is very hard to develop a

working algorith.

- There is no concept of radio power usage, battery level, etc. as discussed in the original design document. I believe these are unnecessary complications and in any case would only be simulated since radios always run on external power. If the aim is to encourage players to conserve power, this can be achieved with appropriate scoring function (e.g. give negative score for excessive number of transmitted packets or high transmission power)
- There is no scoring function defined at the moment.
- There is no backchannel communication between the player's classes implemented. I believe this is an unnecessary complication and current experience shows that it is quite simple to use data in the packet to communicate between the nodes. This is also a more realistic scenario.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index