Wireless Networks In-the-Loop: Speeding up GNU Radio development

Final Report for the Software Defined Radio Design Challenge 2012 at Wireless@Virginia Tech

Gerald Baier and Nico Otterbach
Karlsruhe Institute of Technology, Germany
{gerald.baier, nico.otterbach}@student.kit.edu

***Abstract*—Wireless Networks In-the-Loop can significantly cut the time needed to develop a fully functional wireless communication system. By introducing a virtual RF front-end and a virtual radio channel, which integrate seamlessly with already existing software radio projects, simulations can be run in lieu or in conjunction with real-world tests. The following pages describe a solution and implementation for the GNU Radio SDR framework.**

I. What is Wireless Networks In-the-Loop and why you want to use it

While generally Software Defined Radios are very versatile and allow a quick reconfiguration of devices, making testing far easier compared to conventional radios, testing and the whole development process is still too cumbersome and could be enhanced significantly. One example for a typical wireless system is depicted in figure 1. The whole wireless network consists of multiple nodes, where each node has a reconfigurable software radio part that communicates with the RF hardware.

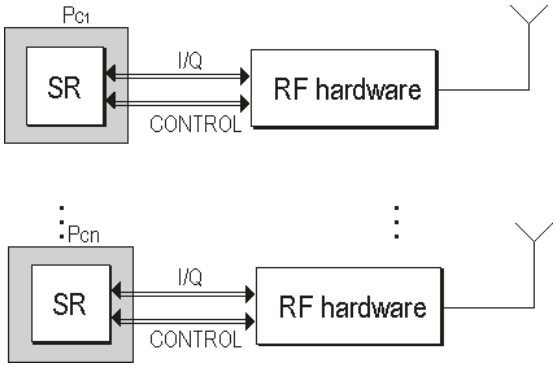


Figure 1. A typical SDR setup [1]

One possible solution to speed up the development of such a system are Wireless Networks In-the-Loop (WiNeLo). The idea behind WiNeLo is quite simple. The RF hardware and the radio channel are being emulated by software. This is shown in figure 2, where the RF hardware has been replaced by a virtual RF interface and the radio channel is being simulated by using a channel matrix. In order to increase the ease of use of the whole setup a central Dispatch and Control Process manages the whole system.

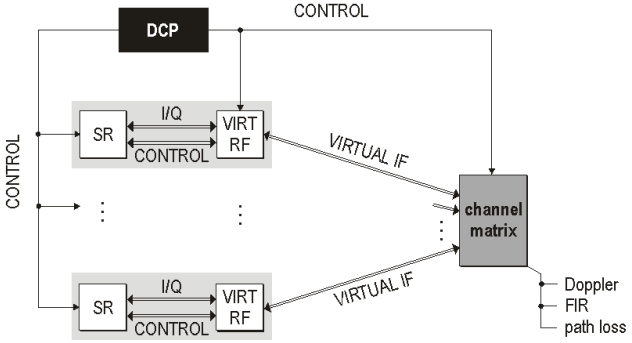


Figure 2. Basic architecture of Wireless Network In-the-Loop [1]

Since the virtual RF interface behaves identically to the real RF Hardware the same software code base can be used for both, the real-world testing (real mode) and a simulation (virtual mode). This is a huge improvement over the conventional development process where adjustments have to be made to both the SDR as to the simulation environment. WiNeLo also enables developers to run cheap tests entirely in software, while a switch to the real mode is possible at every moment. The typical workflow would then consist of running a simulation or a real world test, evaluating the results and making adjustments to the software component of the SDR (see figure 3).

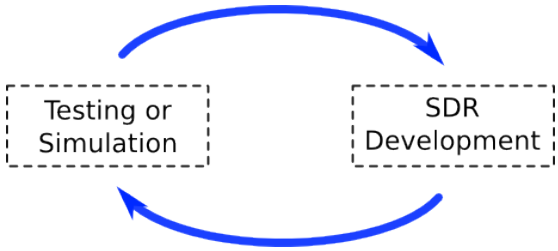


Figure 3. Development cycle with Wireless Networks In-the-Loop

WiNeLo can drastically reduce the time needed for developing a fully functional system, thereby making the lives of developers easier while at the same time cutting costs.

This is one of the many reasons why researchers at our department already worked on a concept of WiNeLo (see [1] and [2]). Two research associates at our department suggested

that we could write an implementation of WiNeLo based on GNU Radio [3]. GNU Radio was chosen as the software radio framework due to the following reasons:

1) Our department has already used GNU Radio in a wide scope, so it is the system with which we are the most familiar with
2) GNU Radio is open source, which makes it possible to alter any aspect of GNU Radio if the need exists
3) There is clean and clear interface between hardware and software, making the replacement of the hardware part easier
4) Its widespread use in academia will make our work relevant and useful for a large number of people

To make the seamless switching between real-mode and virtual mode possible, it is paramount that the actual RF hardware can be replaced with our simulation framework at the push of a button. This ensures that developers can really make use of the coexistence and the synergy between simulation and real world testing.

II. Initial thoughts before kick-off

We have already explained why GNU Radio was chosen. The other thoughts that we had at the beginning of this projects were the following:

In order to distribute the load of all the computation among many computers we had a client-server structure in mind, where the server is in charge of modeling the channels and basic management tasks, while each client is responsible for taking care of one or more nodes of the network.

Since we want to model a complete network with an arbitrary number of nodes, one of the biggest issues is ensuring that all nodes are synchronized and run at the same pace. This is necessary, since the incoming signals from all connected clients have to be overlaid in an exact manner to model the behavior of a real radio channel. If some nodes work at different symbol-rates interpolation is needed, or if nodes do not transmit continuously zeros have to be inserted into the data stream to ensure that the various streams stay coherent. The reason for all nodes to consume and produce samples at the same rate is simply that some nodes may adapt themselves to the channel and the signals on the channels. So we cannot have any nodes producing samples before every other node has finished processing their current batch.

All of the above is directly related to the next relevant point: granularity. Each node may have a certain requirement how fast it needs to react to a changing channel or signals on that channel. The simulation has to take this into account by not working on more than a certain number of samples at any given time. This basically translates into a buffer that the simulation uses with a limited buffer size. Although at the first look this seems to be a serious constraint, it is not that a big a deal, since there is an inherent lag in all GNU Radio flowgraphs due to the buffers between all the signal processing blocks and the GNU Radio scheduler.

Last but not least, we envisioned that the hardware modeling is done by having a distinct profile for an USRP and a specific daughterboard. The long term goal being the modeling of hardware that GNU Radio can use (for example: USRP daughterboards [4], funcube dongle [5] and RTL-SDR [6]). The hardware modeling was to be done either by the server or by the clients, thereby providing an additional possibility to distribute the load among many computers.

This would make it possible to simulate complete wireless networks, regardless of what hardware the transmitters and receivers use and in what kind of propagation environment they are located.

III. Implementation and Results

A. Networking Framework

In order to take care of the client-server structure and connectivity we decided to use the Python network engine Twisted [7] due to the following reasons:

1) It provides all the network functionality that we need and a high level of abstraction, which is a big time saver compared to writing a custom network engine.
2) Twisted is event-driven which makes it useful for us, since we cannot fully control when for example packages arrive or new connections get instantiated.
3) It is written in Python, hence it works quite smoothly in conjunction with GNU Radio.

The biggest obstacle concerning networking was keeping the various clients synchronized and allowing clients to dynamically connect, disconnect and reconnect to the server. Twisted proved to be very useful since it automatically calls certain methods if a specific event occurs. These methods can be easily customized to meet ones needs. We made heavy use of this functionality to cope with dynamically connecting and disconnecting clients during runtime.

B. Client-Server structure

The layout of our client-server structure can be seen in figure 4. All the communication between the clients and the server is being handled by Twisted. Note that in this figure the client is in charge of hardware modeling, but with our framework it is also possible that this is done by the server.

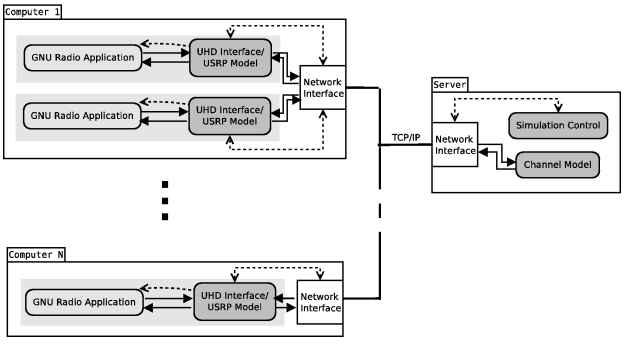


Figure 4. Client Server structure

Since the server is, thanks to Twisted, completely decoupled from the clients' GNU Radio flowgraphs, it does not need

GNU Radio as a dependency, which makes the deployment of the server much easier. The only two relevant dependencies are Numpy and Twisted.

The samples are collected and distributed from and to the clients in the form of packages. The package size itself is set by the server. Every client tells the server the maximum size of a package it can cope with, the server then selects the largest common packet size for transmission. The packet size is directly proportional to the maximum time a client, that needs to react to data on the channel, can tolerate. For example, given a client that has to react within 10ms and if a sample corresponds to 100μs, then the maximum packet size is 100 samples per packet. The largest common size is then used for all transmission.

There is also the gain provided by the load-balancing capabilities of the client-server approach. The GNU radio flowgraphs can be run on multiple machines and especially on a different computer than the one acting as the server. Since all the clients are connected to the server, general management tasks can also be performed by the server.

C. Hardware Emulation

Getting an accurate model of existing hardware platforms for GNU Radio proved to be too complex and time-intensive for the contest. There are just to many variables and varieties you have to take into account like modeling the analog to digital converters, quantization noise or sampling jitter. Hence, we opted to use a generic model which can simulate the following non-idealities

1) Additive white Gaussian noise
2) IQ imbalance
3) Phase noise
4) Frequency offset

As a blueprint for this models we used the generic GNU radio blocks created by Matt Ettus [8]. The hardware emulation can either be done by the server or the clients in order to distribute the load of this computationally expensive task. This has to be set before runtime. However, the parameters of the hardware emulation can be changed on-the-fly, even while a simulation is running. Changing these parameters while a simulation is running could be very useful for engineers to see if their system still works correctly if they environment deteriorates. It is also possible to use this for educational purposes to show students the impact of various types of noise on a constellation diagram.

D. Integration with GNU Radio

As mentioned previously one of our goals was a seamless integration with already existing GNU Radio applications. Therefore we decided that we needed sink/source blocks that behave similarly to UHD sink/source blocks. This is also in accordance with the loop philosophy demanding that switching between simulation and real-world testing should be possible at a moment's notice without hampering the user with tasks like migration etc. For the sink/sources blocks we decided to make use of the newly introduced Python Blocks in GNU Radio. Although the Python block feature is still considered experimental and on average these blocks are slower compared to their C++ equivalents, we used them due to the following reasons:

1) The performance hit should not be too severe since Python blocks make use of numpy which is heavily optimized.
2) Data exchange between the Python blocks and the Python Twisted application should be simple since they are written in the same programming language
3) All the blocks have to do is pass samples from the GNU Radio flowgraph to the Twisted application, no signal processing is involved, so that performance should not be an issue at all.

To further lower the learning curve for prospective users GNU Radio Companion bindings were created. A screenshot of a flowgraph making use of two WiNeLo source blocks can be seen in figure 5. Switching from virtual mode to real mode is done by simply replacing the source blocks with a UHD-source block.

The whole configuration of the WiNeLo-source block is shown in figure 6. The most important options are the server's IP address and its port, the maximum packet size supported by this client and the parameters for the hardware emulation. Callback methods were implemented so that these parameters can be changed on-the-fly from within GNU Radio companion. The client index is used to identify the clients, the client name is only for convenience for users. The option start reactor is required by Twisted. Only the last WiNeLo-source block that is instantiated may start the reactor. Although this requires the user to do this by hand, this was the only solution which does not require custom patches to GNU Radio Companion. The option "HW Emulation on client" sets whether the client or the server is responsible for emulating the hardware. The dialog for WiNeLo-sink blocks is identical.

E. Zero padding and synchronization

For an accurate timing behavior, it is important that the samples of the different clients are overlaid coherently. As the time in the simulation is based on the number of processed samples, a continuous stream of samples is required from every client. Therefore it is necessary to insert zeros into the data stream if a client does not transmit at a certain point in time.

Implementing zero padding was one of the most demanding tasks in the whole project, but in the end a sophisticated and accurate solution was found.

Against our initial considerations for zero padding we finally opted to follow a completely different approach. Recently support for timestamps and stream tags were added to GNU Radio. These tags permit the transmission of data at a preset time. Stream tags are messages that are associated with a specific sample and can evaluated by GNU Radio/UHD blocks. Since the majority of implementations of TDMA/timing critical systems will most likely rely on these methods, we decided to make use of the same functionality.

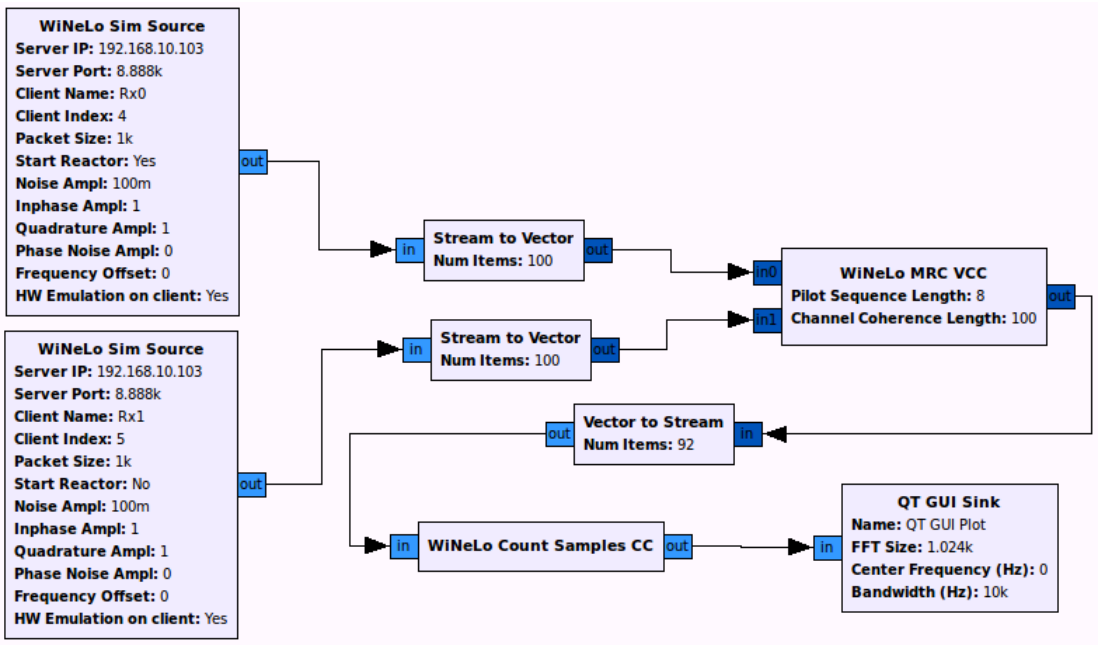


Figure 5. Screenshot of a typical GNU Radio Companion flowgraph, that makes use of two WiNeLo Source Blocks

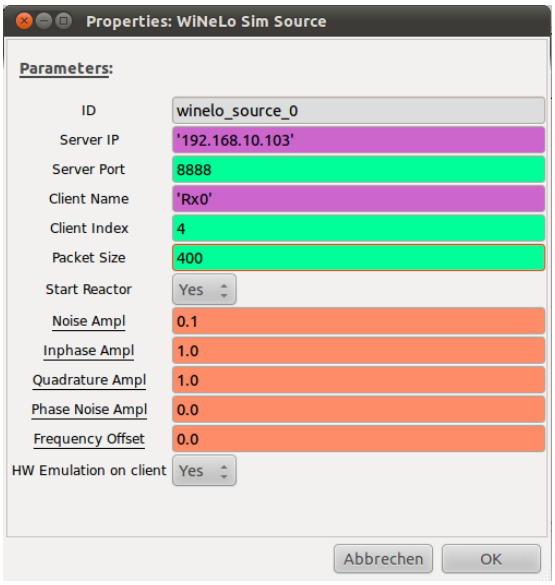


Figure 6. Options of the WiNeLo-source block

Additionally this facilitates using our simulation framework in conjunction with real-world testing, which is fundamental to the seamless switching aspect of WiNeLo.

With our approach no overhead is produced for non-timing critical applications, since the code responsible for zero padding is only executed if needed.

In order to understand our implementation we have to dive a little bit into stream tags in GNU Radio. There are three tags that are of interest to us:

1) SOB - start of burst: Appears whenever a burst begins. When this tag is detected, the USRP switches from IDLE to transmission mode. The SOB tag is usually used together with a timestamp tag that tells the USRP when it should start transmitting. In this case, two tags are associated with one sample.

2) Timestamp: The time carried by timestamps is given in full and fractional seconds. The USRP uses an internal counter to keep track of time. By comparing timestamps to this counter, the USRP is able to transmit at a specific time.

3) EOB - end of burst: This tag is associated with the last sample of a burst and therefore does not need an additional timestamp. It is necessary for the FPGA in the USRP to go to non-transmission mode.

With these tags, their associations to different samples and the knowledge of the sample rate, it is possible to calculate the correct number of zeros that have to be inserted into the stream when a client is not transmitting. The principle is shown in figure 7.

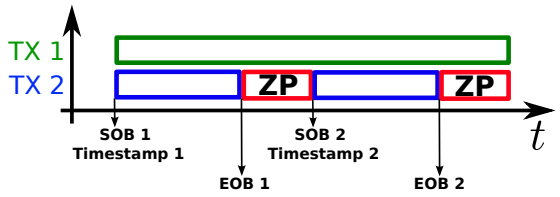


Figure 7. Zero Padding with the help of timestamps

As soon as an EOB tag is detected, we know that zeros have to be inserted. This is done one zero after another, until an SOB together with a timestamp tag is detected (this marks

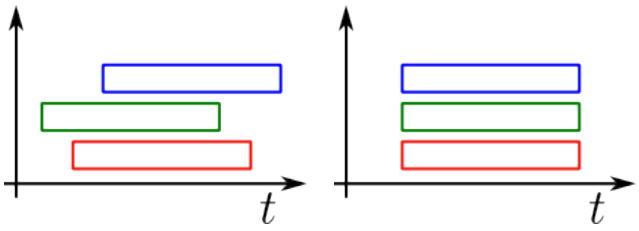
the end of the sequence of zeros).

After the detection of a new SOB timestamp the equivalent offset in samples regarding the start of the simulation is calculated. This is done with the help of the used sample rate, which is necessary to convert the time given in seconds from the timestamp into an offset of samples that will be used by our simulation. The total number of zeros to be inserted is now given by the just mentioned offset minus the number of already processed samples since the simulation is running (this is our elapsed virtual time, the elapsed time in our simulated world and the equivalent to the internal counter/clock in the USRP).

Thus, we are able to set up our virtual time base with a maximum error of only one sample compared to real-world behavior.

Now, since all of the clients run at the same pace and produce new samples at the same rate, they share a common clock. With this new virtual time base it is possible to simulate real-time systems in non-real-time.

The principle we used for synchronization is shown in figure 8. The server overlays the streams sample by sample after it has received a full package from every client.



(a) (b)

Figure 8. (a) unaligned packages received by the server; (b) packages after synchronisation at the server

With our implementations of synchronization and zero padding, it is now possible to simulate TDMA/real-time systems with a very high accuracy and in non-real-time.

F. Simulating the channel

The simulation of the channel itself is performed by the server, which takes the signal from every transmitter that reaches a certain receiver, applies the corresponding channel impulse responses and does a summation of all signal to get the total received signal. Due to the delay caused by the channel impulse response, some samples of the previous packages are needed for computing the signals at the receivers. If we are dealing with a channel that changes with time, the server is also in charge of switching the channel matrix. All of these operations are done with the help of the Python module Numpy, which resulted in a decent performance but is probably still significantly slower than an implementation in C++ or Fortran.

So far there are two possible ways how the server can deal with connecting clients.

1) Dynamic Mode: In the dynamic mode the server accepts connections from an arbitrary number of clients. The channel matrix is completely random (the attenuation is Rayleigh-distributed) and its size is automatically adjusted to the number of connected clients. This is made possible by the event-driven nature of Twisted, which makes it very easy to run certain methods and functions as soon as a client connects or disconnects. This mode can be used for quick trial and error tests, due to the fact that it hardly requires any configuration on the server side. The server can be configured to generate a new channel matrix after a certain amount of samples. This can be used to simulate fading in a time-variant channel.

2) Fixed Mode: In this mode the server loads a channel matrix from file and waits until the appropriate number of clients has connected. As soon as all clients have connected the simulation starts. As in the dynamic case the server can load a new channel matrix after N samples. The channel matrix is a numpy array or a list of numpy arrays. This will make it easy to use the simulation framework in conjunction with channel impulse response obtained through measurements or try out custom channel models (two-way propagation, AWGN, ...).

IV. Further Work / Outlook

Due to the limited time that we had on our hands and the vast scope of a project like a complete simulation framework, there still remain a lot of things to be done:

- Implementing more channel models would increase the scope in which the simulation framework can be used effectively
- The hardware modeling is still very basic and in an early stage of development. What we imagine is to have a separate model for every hardware that is available for GNU Radio.
- Currently the server only works on one packet at a time. We think that the network performance could be increased by introducing queueing for the packages
- Implementing the channel matrix itself as a GNU Radio flowgraphs would increase performance significantly and would allow the use of custom GNU Radio blocks as a channel model.
- Adding support for different sample rates at the clients.
- Using advanced features of Twisted could increase network throughput.
- Converting the Python Blocks to C++ Blocks would eliminate the need to use the experimental branch of GNU Radio (Josh Blum's repository).
- Support for absolute timestamps (simulation equivalent for using a GPSDO with USRPs)

V. Conclusion

The last pages illustrated the usefulness and the general principle of Wireless Networks In-the-Loop. The simulation framework that we have created from scratch over the last 6 months (both of us spent roughly 40 hours per month on this project), while still being work in progress, already shows some of the benefits from WiNeLo.

We also feel that some kind of distributed computing is imperative, in order to cope with networks of larger size. In our framework this is accomplished by using Twisted, which allows us to treat the network nodes and the server independently on different machines. The next step would be to distribute the actual simulation of the channel among many computers, which should not pose much of a problem since all the receive paths can be computed independently from each other. Additionally since Twisted is extremely versatile, there are hardly any limits on how you can use it. We have already ran a simulation over an SSH tunnel and are currently thinking about hosting the server with the help of a cloud computing service like Amazon Elastic Compute Cloud (Amazon EC2).

In general, our simulation framework permits users to simulate real-time systems in non-real-time. Switching between simulation and testing requires hardly any effort thanks to the WiNeLo sink/source blocks and their GNU Radio companion bindings. Furthermore the capability of the framework to cope with dynamically connecting and disconnecting clients is, at least to our knowledge, unique. All of this feature should prove useful for a developer and provide him with a significant advantage over conventional development methods.

REFERENCES

[1] J. Elsner, M. Braun, S. Nagel, K. Nagaraj and F. K. Jondral, "Wireless Networks In-the-Loop: Software Radio as the Enabler", in *Software Defined Radio Forum Technical Conference*, 2009.

[2] S. Koslowski, M. Braun, J. Elsner and F. K. Jondral, "Wireless Networks In-the-Loop: Emulating an RF front-end in GNU Radio", in *SDR Forum 2010 European Reconfigurable Radio Technologies Workshop*, 2010.

[3] Free Software Foundation, "GNU Radio", http://www.gnuradio.org

[4] M. Ettus, "Ettus Research LLC", http://www.ettus.com

[5] Funcube Dongle, http://www.funcubedongle.com/

[6] RTL-SDR, http://sdr.osmocom.org/trac/wiki/rtl-sdr

[7] Twisted, http://twistedmatrix.com

[8] M. Ettus, "Practical Software Radio: why things don't always match the textbooks", http://gnuradio.org/redmine/attachments/download/249/02-ettus-practical-radios.pdf