GWN: A FRAMEWORK FOR PACKET RADIO AND MEDIUM ACCESS CONTROL IN GNU RADIO

Víctor Gonzalez-Barbone (vagonbar@fing.edu.uy)[1], Pablo Belzarena (belza@fing.edu.uy)[1], Federico Larroca (flarroca@fing.edu.uy)[1], Martín Randall (mrandall@fing.edu.uy)[1], Paola Romero (paolar@fing.edu.uy)[1], and Mariana Gelós (mariana.gelos@fing.edu.uy)[1]

[1]Facultad de Ingeniería, Universidad de la República, Montevideo , Uruguay

ABSTRACT

Software Defined Radio, and GNU Radio in particular, were conceived for communication systems such as radio and TV, where information is conveyed in a continuous flow, called a stream. Data networks by contrast use small portions of information, called messages, frames or packets according to the context. A further important difference is that in data networks several nodes share the same medium. In order to operate properly, a so-called medium access control must be enforced.

GNU Radio was originally stream oriented, but more recently added support for message communications. A block may thus comprise two different types of inputs and outputs: stream ports for continuous flows of data, and message ports for discrete portions of bytes. As a consequence, some projects that strive at implementing data network standards in GNU Radio have emerged, but they are oriented towards specific communication protocols.

In this paper we present GWN (GNU Radio Wireless Network), an open and free extension to GNU Radio specifically oriented to data networks, but not tied to any specific protocol. Its aim is to provide a framework for experimentation and development, working either on existing protocols or devising entirely new ones. To this purpose, GWN provides a new generic block (`gwnblock`) which adds the tools necessary for data network designs, and at the same time decouples all GWN data network blocks from the GNU Radio generic basic-block. This means a new GWN block only needs to inherit from `gwnblock` and follow GWN design rules to build a data network application, while keeping full compatibility and access to GNU Radio standard blocks.

The GWN generic block adds the following facilities to GNU Radio: Message orientation, Events, Handling of time, and Finite State Machines. As a proof of concept, and to illustrate its usage, we briefly present two examples: an ARQ (Automatic Repeat reQuest) protocol with its different flavors and a CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) protocol.

1. INTRODUCTION

The ultimate objective of Software Defined Radio (SDR) is to implement a complete communication system in software. Current SDR implementations allow to sample a portion of several megahertz of the spectrum and feed them into a programmable device (such as a personal computer), where a software suite processes them as required. Transmission is analogous. This allows to implement a cellular base station [1], an FM radio [2], or a Digital TV receiver [3] by running different programs on the computer, just to name a few examples.

Instead of using expensive proprietary hardware implementations, SDR allows to design and implement most of the communication system in software, which may be tailored specifically for the given application. Several SDR software suites are free and open, the most popular being GNU Radio [4]. Moreover, SDR hardware implementations also enforce some level of openness. Open-source technologies transcend geographical and cultural boundaries, as anyone in any place can contribute and collaborate. This new paradigm offers new opportunities for research and teaching in telecommunications, providing a freely available design framework apt for technical education, research, development, and deployment, through a cost-effective, easily approachable tool with modest infrastructure requirements.

GNU Radio was originally designed to support the processing of continuous data streams from a source to a sink passing through different blocks, each performing a specific task (e.g. filtering, decision). The stream is a flow of basic types like bytes, integers or complexes. Each GNU Radio block defines input and output signatures which specify the number of input and output streams and their respective type. The designer can choose which blocks are needed and how they are connected to build a flow graph for a particular physical layer implementation. GNU Radio has an internal scheduler which invokes sequentially each block and communication between blocks is performed through shared memory.

This data stream model works well for samples, bits, etc., but it is not appropriate to handle control data, metadata, and packet structures. GNU Radio partially alleviates this problem by introducing two new communication mechanisms. The *tag* system is a stream parallel to the data stream that holds metadata and

control information. This mechanism allows to add additional information to a particular sample or flow, but the paradigm is still the same. A *message passing* system was also added, with two main goals: to allow downstream blocks to communicate back to upstream blocks, and to provide an easier way to communicate between external applications and GNU Radio.

As we discuss with greater detail in the following section, further extensions are necessary for GNU Radio to support packet communication, in particular handling of time and support for finite state machines (FSM), a preferred way to implement data network protocols. Thus, the main goal of our research is to provide a fully functional SDR-based wireless network, developed on top and integrated with GNU Radio. To date, we have defined a general framework which allows the implementation and use of different data link protocols (and as a consequence wireless networks) integrated to the GNU Radio project, which we have thus named *GNU Radio Wireless Networks* (GWN). This framework is being developed as a free, open source software project under the GNU license, with the explicit intent of disseminating these ideas, contribute our present achievements, and allow interested researchers and developers to contribute.

This article is structured as follows. First, we analyze the Data Link Layer protocols requirements. Based on these requirements we describe the architecture of GWN and we explain the methodology used to build the GWN framework. Next, we analyze how to use and how to extend GWN. Finally, we explain how GWN has been tested and we state the results and conclusions of this work.

2. DATA LINK LAYER REQUIREMENTS

GNU Radio and more generally SDR stem from radio frequency communications such as radio and TV. In these fields, information is conveyed in a continuous flow, called a stream. On the other hand, data networks use messages, frames or packets according to the context. A file transfer is thus carried out by dividing the mass of bytes in small packets which travel through the air on their own. Several distortions may affect their travel: their delay may be different and arrive out of order, they may be altered in their content, they may be lost and require retransmission. At the receiving end, these packets must be validated for errors and either corrected or asked for retransmission, ensure all of them have arrived, restore the correct sequencing, and then be aggregated to rebuild the file exactly as the original.

Moreover, in radio and TV the electromagnetic spectrum is divided into frequency bands, called channels, to avoid mutual interference. In bilateral radio communications either transmission and reception must happen in different bands, or the parts must take turns to speak. In data networks many actors share the same communication channel, called a shared medium. Some discipline must be imposed to avoid "all speaking at the same time". This discipline is called a *channel access method*.

All in all, data networks involve multiple users on a shared medium, use packets, these packets may suffer errors, variable delays, losses, and sequence alterations. These and other problems are addressed in a number of standard which regulate network communication, from small local area networks to the Internet.

At data link layer, each protocol control logic is typically based on a state machine that implements a set of service primitives. State machines are very effective in modeling the behavior of sequential control operations, and most MAC protocols and other link layer protocols are formally described in terms of state machines [5].

The state machine reacts to different actions performing state changes and/or generating a new set of actions. These actions may handle control or management data units, but also modify or reconfigure this block or other blocks. These actions and service primitives can be implemented as asynchronous events. This is another main requirement of data link layer protocols: the communications between blocks must be driven by events. A block may also require a service primitive from another network layer. For example a CSMA/CA block will need to ask the physical layer for the channel state (if it is idle or not).

In addition to the above, there is another important type of event that must be handled at data link layer: timer events. For example in CSMA/CA or in ARQ protocols each time a packet is sent, the control logic of the protocol must start a timer. When the timer ends, it generates an event informing the control logic of its expiration. If a packet with an acknowledgment arrives after the timer expiration the packet must be resent, and if it arrives before the timer must be stopped. Therefore, data link layer protocols require to handle a set of asynchronous timers.

GNU Radio was originally stream oriented, but as we explained in Sec. 1., it recently added support for message communications. A block may thus comprise two different types of inputs and outputs: stream ports for continuous flows of data, and message ports for discrete portions of bytes. This is an important step to allow data link layer protocols implementation in GNU Radio. However, as discussed before, it is also necessary to associate a finite state machine to each control block, as well as the capacity of handling events and (several) timers also associated to each control block.

Some projects exist to implement data network standards in GNU Radio (for instance [6]), but they are oriented towards partial implementations of specific communication protocols. We developed GWN as an extension of the GNU Radio toolkit specifically oriented to data networks, but not tied to any specific protocol; its aim is to provide a framework with the tools for experimentation and development, working either on existing protocols or devising entirely new ones.

Summarizing, a framework for implementing data link layer protocols must be modular, flexible and adaptable, not only to allow modification or replacement of a certain protocol but also to seamlessly include new and future network architectures with a moderate effort. This framework must include the capabilities to easily implement finite state machines and timers, as well as the ability to handle different types of events.

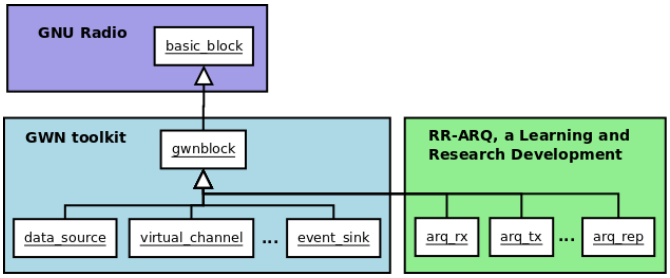

Figure 1: GWN blocks.The GWN toolkit provides a generic `gwnblock` which inherits from GNU Radio *basic_block*. All GWN toolkit blocks and blocks developed by students and researchers inherit from `gwnblock`. RR-ARQ is an example implementation of Automatic Repeat reQuest (ARQ), an error control protocol for data transmission.

In view of the success achieved by GNU Radio, our approach was to follow its design model and to add functionalities to GNU Radio blocks. This is discussed in the following section together with an introductory description of GWN.

3. GWN ARCHITECTURE

GWN extends GNU Radio towards data networks in a toolkit with its own features. Blocks in GWN and GNU Radio can be mixed in the same flowgraph. New GWN blocks can be built alongside the GWN design in the certainty of their compatibility with both GWN and GNU Radio. To this purpose, GWN provides a generic `gwnblock` which adds the tools necessary for data network designs, and at the same time decouples all GWN data network blocks from the GNU Radio generic `basic-block`. This means a new GWN block only needs to inherit from `gwnblock` and follow GWN design rules, shielding from users most of the complexity of GNU Radio.

In addition to the GWN generic block, the GWN toolkit includes some common function blocks such as message sources and sinks, a channel emulator, message converters, and framers. This allows for the demonstration of basic network data communications just by interconnecting GWN blocks in a flowgraph. Figure 1 shows how learning and research developments need only interact with GWN in their construction. This architectural scheme simplifies access of students to development, both in coding and in documentation. In this sense, special care has been taken to provide complete, readable documentation on all GWN blocks and tutorial material on new block development.

In particular, the GWN generic block adds the following facilities to GNU Radio.

1. Message orientation. GNU Radio is mainly stream oriented, GWN is message oriented; items interchanged among GWN blocks are discrete groups of bytes. GWN makes use of the message mechanism of GNU Radio, but provides some blocks to interact with stream GR blocks when necessary, thus relieving users of stream oriented worries.
2. Events. GWN elaborates on the message interchange mechanism of GR into a more structured item of interchange called an *event*. GWN blocks interchange events. The event inner structure reflects the needs of network data protocols and is closer to their design conception.
3. Handling of time. This is a feature absent in GNU Radio, and essential in data networking. Answers are waited for a certain time, keep-alive signals are emitted at regular intervals; timing pervades data communications. GWN provides two forms of handling time: timeouts and timers. A timeout just waits for some time and emits a timeout event; it is a one-shot gun. A timer emits timing events regularly.
4. Finite State Machines. Most data communication protocols involve a complex logic usually described in a mathematical model of computation called a Finite State Machine (FSM). An FSM comprises *states* and *transitions*, and reacts to events: when the machine is in a certain state and receives an event, a transition to another state is performed, optionally with some parallel task. FSMs are a very powerful tool, and the complexity of some protocols makes it almost impossible to implement them otherwise.

All GWN blocks communicate among themselves by the interchange of events. An event is an instantiation of the GWN Event class, described by a nickname, and including a type, a subtype, and optionally other items in dictionary form. The GWN Event class is intended to be subclassed in a hierarchy of different event types. This allows to define event classes which closely reflect the contents of the different types of packets used in data networks, were they control, management or data. Besides, events can also be used to interchange information among blocks, such as the timer events used to start a timer, signal a timeout, or perform some action at regular periods for a number of times.

The inner structure of a typical GWN block is shown in Figure 2. Messages are received and sent as PMTs (Polymorphic Data Type), the standard data type in GNU Radio. From these messages the encapsulated events are recovered and passed to the `process_data` function, the only place where a programmer must code the functions of the block. This function also receives events from timers and timeouts, and from the FSM. According to the events received, and the logics of the function to be performed, a new event is generated and sent to the output ports. Encapsulation and recovery of events from PMT messages is done in the input and output ports in a way transparent to the programmer, who may just think in "events".

To create a new block, the user indicates the number of input and output ports, and codes the logic in `process_data`. Optionally, the number of timers and timeouts may be indicated, and an FSM may be loaded with transition rules. This is a quite straightforward process: inheritance from `gwnblock` ensures ports, timers, timeouts and FSM work as expected. The use of the PMT data type to interchange among blocks makes GWN blocks fully compatible with GNU Radio blocks.

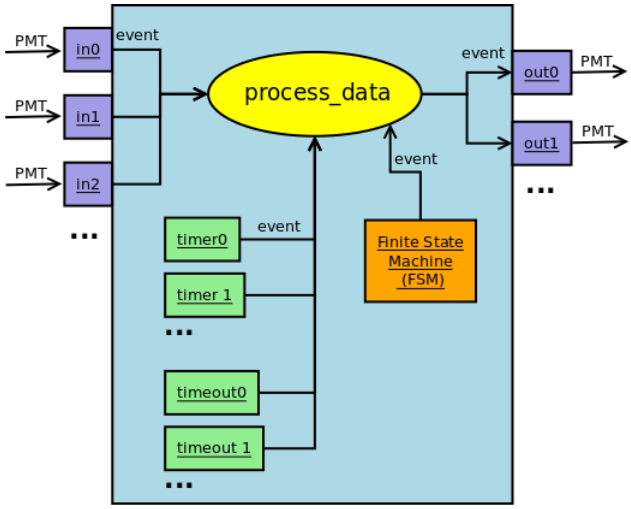

Figure 2: A typical GWN block. Messages (PMT) received on input ports are decoded into events and served to the `process_data` function, which also receives events from timers, timeouts and the FSM. The `process_data` function generates and emits events through the output ports as messages (PMT).

GWN includes a simplified version of an eXtended Finite State Machine (XFSM) which has been used to implement complex packet processing tasks inside network switches, and is considered a powerful enough tool to implement any protocol for data networks [5].

The framework is built on a modular architecture of building blocks, where each block performs some specific functions. These blocks communicate with each other according to protocols implemented as asynchronous message events. This concept allows for the integration of several blocks to form a new block capable or performing a set of related tasks. Thus, it becomes relatively simple to integrate functionalities implemented in different blocks to cover the different requirements of a wireless network, such as medium access, neighbors discovering, etc. The GWN code and documentation may be found in [7].

The next section describes the methodology of research and design used in this project, where one of the most difficult challenges is to evolve from sheer prototyping into sound architectural design.

4. METHODOLOGY

In the area of telecommunications, as well as in many other technical areas, research is based on extensive prototyping, followed by testing, correcting, and further prototyping. In this way, many proposals in the area end up with a weak design in software architecture. Since considerable work has been done, and positive results have been obtained, architectural design naturally falls to a second plane. In many cases, once the desired results have been obtained, and the final prototype is working as desired, there is not much motivation to worry about software architecture, so much so that it would imply to almost start all over again. This is not the case with this project: a far reaching enterprise as this calls for a reasonable architectural design, well modularized, with clear interfaces, consistent patterns of development, and quality documentation, so as to make it apt for reuse and extension.

The first stage of this project was mainly prototyping, with a rather flat package and module organization; its purpose was the development and testing of software modules to implement a wireless network link in software over a generic piece of hardware (e.g. USRP) using GNU Radio for the physical layer. Once this goal was achieved, some architectural requirements specific of the project started to emerge. An analysis and evaluation of existing architectural designs and patterns was carried out, looking for the best approach towards an architectural design of the framework. The GNU Radio model was also studied and evaluated.

The convergence of prototyping and architectural design evaluations led to a first approach in the architectural design of the GWN framework. Needless to say, the first draft of a software architecture, in a far reaching enterprise as this, is most critical. At the same time, it is very difficult to envision all possible architectural requirements at an early stage. For this reason, our methodological approach was conceived as a balance between prototyping and design, in which design tries to apply good software engineering practices to the needs discovered by prototype work, and prototypes test the usability of the design, show its shortcomings, and help improve design.

5. FIRST STEPS USING GWN

5.1. A working example

GWN allows a step by step construction, ideal for showing and experimenting how each block performs its duty. The simplest flowgraph is an event source connected to an event sink: events produced in the first block are displayed by the second. From this on, gradual addition of blocks may lead to the simulation of a transmission over air, using a channel emulator block in place of hardware and air.

Experimenting a communication link with emitter and sender in the same personal computer, as described above, is a must task for beginners. Once the simulation works, substituting the channel emulator by the SDR hardware, one for emission and the other for reception, should render a working, real communications link. In this way two computers can be interconnected, and start a chat session, a file transfer, or a graphical application in the remote machine. Figure 3 shows the flowgraph to establish a bilateral link with another node.

Along the former lines, data network protocols can be implemented, tested, and improved, starting from simulation and ending in real world communications. This brings a hands-on experience on the many difficulties data network communications face, and the effectiveness of protocols to achieve reliable

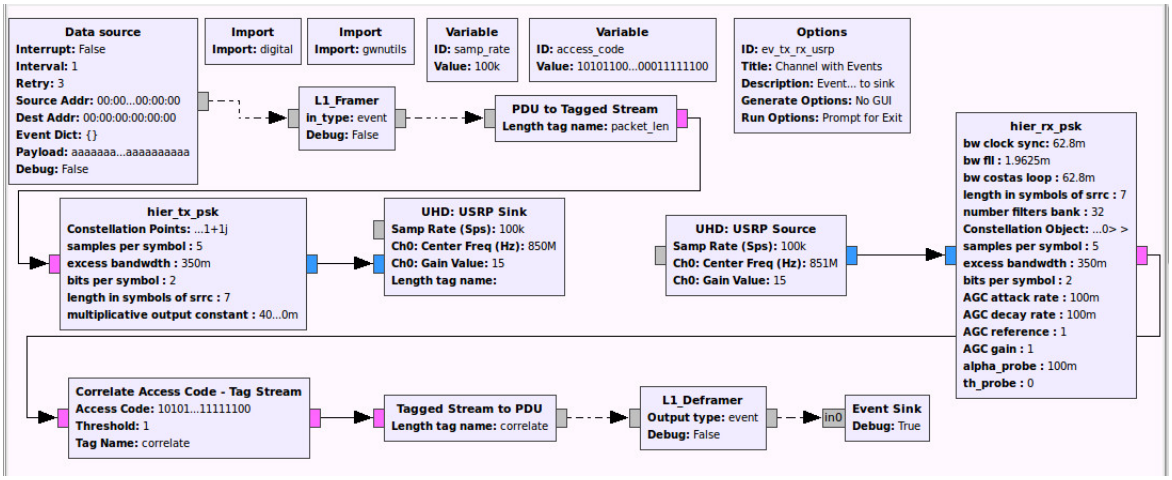

Figure 3: A flowgraph to establish a bilateral link with another node. The `Data source` block produces events at regular intervals. Event objects are serialized and packed as PDUs (Protocol Data Units) in our block `L1_Framer`, which are transformed into tagged streams by GNU Radio's `PDU to Tagged Stream` block. These are modulated into a complex signal and sent to the `UHD: USRP Sink`; this block interacts with the RF device, which sends the signal over the air. At the same time, this same node is receiving messages through the `UHD: USRP Source`, which captures the signal from the air. The received stream of bytes is correlated to detect an access code in `Correlate Access Code - Tag Stream`, and the resulting tagged stream is fed into `Tagged Stream to PDU`, and `L1_Deframer` recovers the original event produced by the sender. Please note that all blocks present in the flowgraph are included in GNU Radio or GWN.

data transfer among a group of nodes, i.e. computers with RF peripherals.

5.2. Building a new block

GWN can be extended by creating new blocks to implement new functionalities. GWN has been coded in Python, and can be extended using the same language. Though in the future some GWN blocks are expected to be rewritten in C++ for performance reasons, extensions in Python will always be possible, as in GNU Radio. This section describes how to create a new GWN block by means of a simple example. In what follows, we create a block called `Virtual Channel`, a block we mentioned at the beginning of this section. In particular, this block receives an item, generates a uniform random number in the interval $[0, 1]$, and writes out the item received only if the random number is greater than a probability of loss set as a parameter of the block. The block has one input, and one output.

After a new block is created (for which GNU Radio's `gr_modtool` may be used) we first add the following imports to the code of the block:

```
from gwnblock import gwnblock
import pmt
```

Next, we modify the class definition to inherit from `gwnblock`, the GWN generic block including implementations for message ports and internal timers, and write its constructor according to the functionality described before. That is to say:

```
class virtual_channel(gwnblock):
    def __init__(self, blkname='virtual_chanel
        ', blkid='id_virtual_channel',
        prob_loss=0):
            gwnblock.__init__(self, blkname=
                blkname, blkid=blkid,
                number_in=1, number_out=1,
                number_timers=0)
                    self.prob_loss = prob_loss
                    return
```

Now we write the processing function (i.e. `process_data`), which in this case will generate a random number and write out the received message only if it is greater than the parameter `prob_loss`:

```
def process_data(self, ev):
        rand_nr = random.random()
        if rand_nr <= self.prob_loss:
                pass
        else:
                self.write_out(ev)
        return
```

GNU Radio provides a graphical interface to build flowgraphs interconnecting blocks; this is called the GNU Radio Companion, or GRC for short. To make the new block available in GRC, an XML file describing the block's ports and parameters is required. A template version for the XML file is created by the `gr_modtool` script, and must be updated as usual in GNU Radio.

6. GWN IMPLEMENTATION OF A FSM

This section describes the GWN implementation of a Finite State Machine (FSM). An instance of the GWN FSM can be associated to a GWN block to implement its logic.

In addition to the usual states and transitions, GWN's FSM includes *actions*, *memory*, and *conditions*:

- An action is a user-written function executed on a transition, before setting the machine to the next state.
- Memory may be any object capable of recording and retrieving information, in whatever access mode the application may need (LIFO, FIFO, etc). The memory facility is not part of the FSM machine, but an independent object. Memory may be handled in the action functions.
- A condition is another user-written function or expression which produces True or False when executed or evaluated. The action function and the transition are only executed if the condition evaluates to True. If the condition evaluates to False, no action is executed and the machine remains in its current state.

The FSM is defined through tables of transitions. For a given input (termed *symbol* in this context) the `process()` method of the FSM uses these tables to decide which action to call and which the next state will be, if and only if the condition evaluates to True, otherwise nothing happens.

The table of transitions defines the associations `(input_symbol, current_state) -> (action, next_state, condition)`, where `action` is a function, `symbol` and `state` can be any object, and `condition` is a function or an expression which returns a boolean. This table is maintained through the FSM methods `add_transition()` and `add_transition_list()`.

In the following example we show the definition in the code of a transition:

```
fsm.add_transition ('goA', 'INIT', fn_goA,
    'State A', "self.where=='A'")
```

This code adds a transition to the FSM `fsm` where if the input symbol is `goA`, the FSM is in state `INIT` and the variable `self.where` is equal to `'A'`, then the function `fn_goA` is invoked and the FSM moves to the state `State A`.

Transitions valid for any input symbol may also be defined. That is to say, associations of the kind `(current_state) -> (action, next_state, condition)`. This table is maintained through the FSM method `add_transition_any()`.

Finally, the FSM may also have a default transition not associated with any specific input symbol or state. The default transition matches any symbol on any state, and may be used as a catch-all transition. The default transition is set through the `set_default_transition()` method. There can be only one default transition.

Thus, upon receiving a symbol, the FSM will look in the transition tables in the following order: 1. The transitions table for `(input_symbol, current_state)`. 2. The transitions table for `(current_state)`, valid for and any input symbol. 3. The default transition. 4. If no valid transition is found, the FSM will raise an exception.

Matched transitions with the former criteria may produce a list of `(action, next_state, condition)`. The condition is evaluated for each tuple in the list, and the first tuple on which the condition is found True is executed. That is to say, the corresponding action function is called, and the next state is set as the current state. If no transition is defined for an input symbol, the FSM will raise an exception. This can be prevented by defining a default transition. The action function receives a reference to the FSM as a parameter, hence the action function has access to all attributes in the FSM, such as `current_state`, `input_symbol` or `memory`. The GWN Finite State Machine implementation is an extension of Noah Spurrier's FSM [8].

7. GWN TIMERS

This section describes how to add timing to GWN blocks. GWN internal timers are objects that can be attached to any block, in the same way as input or output ports. The number of timers to create in a block can be indicated as a parameter in the invocation to the `gwnblock` constructor as shown in Sec. 5.2. with the Virtual Channel example. Let us recall that all GWN blocks inherit from GWN's basic block `gwnblock`.

In its present version, GWN provides two different mechanisms for timing: GWN Timers and GWN Timeouts. Both act sending messages to the block to which they are attached. Messages from timing blocks are made available in the block's `process_data()` function, as if they had been received at an input port. Timing messages can be recognized by their type, i.e. a timing message must be in some way different from messages received at any input port.

7.1. Using GWN Timers

A GWN Timer sends a first type of message to the block to which it is attached to at regular intervals, for a given number of times. Then it sends a single second type of message to indicate the first series has exhausted. A GWN Timer accepts the following parameters:

- `nickname1`: message to send at regular intervals for `retry` times.
- `interval`: time between messages.
- `retry`: how many times to send message 1, then send message 2 once.
- `nickname2`: message to send when retries have exhausted.
- `interrupt`: if True, the timer is interrupted, i.e. it does not send any messages, but keeps alive, and the `retry` counter

keeps counting; when set to False, sending of messages is restored.

A GWN Timer can be controlled through the following functions:

- `set_interrupt(True | False)`: sets interrupt state.
- `stop()`: stops the timer and no more messages will be sent.
- `reset()`: sets the counter to 0, interrupt to False, and starts counting again and sending messages.

The following excerpts of code show the use of internal timers in a block called `Timer Source`. This is a simple block which produces messages regularly as 'timer events'. These messages can be used by other blocks to trigger some action.

This block uses an internal timer to produce a certain type of event regularly with a given interval (the `interval` parameter), for a specified number of times (the `retry` parameter). Events produced are of the type defined by the parameter `nickname1`. Once the retry number has exhausted, a final event of the type defined by the parameter `nickname2` is written out.

```
class timer_source(gwnblock):
'''Timer events source, sends Events
    produced by an internal timer.

def __init__(self, blkname='timer_source
    ', blkid='timer_source_id', interrupt=
    False, interval=1.0, retry=5,
    nickname1='TimerTOR1', nickname2='
    TimerTOR2'):
     # invocation of ancestor constructor
         gwnblock.__init__(self, blkname,
             blkid,number_in=0, number_out
             =1, number_timers=1)
         self.counter = 1  # counts until
             retry
         self.time_init = time.time()  #
             retuns current time
         self.set_timer(0, interrupt=
             interrupt, interval=interval,
             retry=retry, nickname1=
             nickname1, nickname2=nickname2
             )
         self.start_timers()
         return
```

The timer events produced by this block are received by this same block as any received event, hence the function `process_data()` should be written to handle this type of event, as any other. The type of event can be determined within the function to act accordingly.

```
def process_data(self, ev):
        '''Sends timer events produced by
            the internal timer.
        @param ev: an Event object.
        '''
        ev.frmpkt = 'Timer Event ' + str(
            self.counter)
        self.counter += 1
        self.write_out(ev, port_nr=0)
        return
```

7.2. Using GWN Timeout

A GWN Timeout object sends a message after some specified interval has elapsed. It can be interrupted before its action starts, and hence no message will be received. It can also be restarted in a new cycle. A GWN Timeout accepts the following parameters:

- `timeout`: the time before message is sent.
- `nickname`: message to send.

A GWN Timeout can be controlled through the following functions:

- `start(timeout=None, nickname=None)`: starts timeout counting.
- `cancel()`: stops counting. If `timeout` has not been reached no message will be sent.

A note on efficiency. A GWN Timeout may be more computer efficient than a GWN Timer, since a GWN Timer runs in its own thread, which cannot be destroyed without destroying the GWN Timer object. On the other hand, the GWN Timeout creates a Python `threading.Timer` object, which runs in its own thread, and is destroyed on invocation of the GWN Timeot `cancel()` function. This is more accurate, since the `threading.Timer` object is immediately canceled, without waiting for object destruction (creation and destruction of objects may be demanding in process time).

8. GWN AS TEACHING TOOLS

Our goal in the development of GWN is to build a framework that can be usable for education, experimentation, and research in wireless networks. In this section we explain how we have used GWN as teaching tool. Our integration of GWN into Education was carried out along these trends. First, use of GWN as such, in demonstration of data oriented communication and the problems involved, e.g. losses and corruption of messages, both in simulation and real word communications, by implementing a data link between two personal computers. Second, extension of GWN, adding blocks for new functionalities. The goal here was twofold: the implementation per se and the early training of students in research.

Moreover, documentation of blocks is quite complete, tutorial information is given in the project's wiki [9], example flowgraphs can be found in the examples subdirectory after installation, and also on the project's homepage [7]. In this version

GWN is coded in Python, and extensions can be also written in Python. This makes code more accessible to students working on extension projects. A migration to C++ is expected to occur in the near future, for performance reasons, but it will only affect `gwnblock` and associate classes, which are maintained by the GWN team; student developments will not be affected. As to the fast renewal of versions in GNU Radio, extensions of GWN are shielded from them by `gwnblock`, because all GWN blocks inherit from it.

Though GWN is used for demonstrations in the classroom, it excels in the lab, where students can interact freely with it. Educational purposes pursued include demonstration and experimentation in data networks, where students can see "in action" what they learned in introductory courses on data networks. Furthermore, we make use of the toolkit for class assignments and small projects, using the available blocks or simple adaptations. The latter requires some basic knowledge of Python, but the structure of GWN blocks allows to include code only in the `process_data` function; the task is easily achieved with some guidance. Finally, we have used the framework for end of course projects, graduation projects in telecommunications engineering, research, and research training, both in undergraduate and graduate levels.

In the following section we describe two application projects: students implemented ARQ and a CSMA/CA protocol, as an extension to GWN. This projects required the use of timing and the implementation of Finite State Machines. The following section illustrates the use of these two features of GWN.

9. TESTING THE FRAMEWORK

Automatic Repeat reQuest protocols are layer 2 control protocols for ensuring packet exchange. Even though there are different versions ranging from simplicity to efficiency, they all share some common premises: packets are to be acknowledged (and otherwise considered lost), and packets are to be delivered in proper order to their destination.

In order to test GWN as a simple but complete framework for communications protocols, the task of developing the ARQ protocols was conveyed to undergraduate students. With no prior knowledge of GNU Radio programming, and using the tutorial provided by the GWN team, three variants were implemented and tested (Stop and Wait, Go Back N and Selective Repeat), both in GRC simulation flowgraphs and real wireless communication. A second programming of the protocols was done using FSM, with their respective tests.

Finally, as part of a graduate project in telecommunications engineering, a CSMA/CA protocol was implemented using FSM. It was also tested in both simulation and real life communications, implementing a simple chat application.

These protocols were implemented in two GWN blocks: one in the transmission node with the logic of the protocol, and the other in reception mainly for acknowledgement. In the next sections we discus the Selective Repeat and CSMA/CA protocols as we believe them to be the most illustrative.

9.1. ARQ: Selective Repeat in FSM

The Selective Repeat protocol is the most complex ARQ protocol as it implies some data processing at reception. The transmitter uses a sliding window for sending packets and each packet has a timeout of its own, which starts when the packet is sent. When an acknowledge (Ack) for this packet is received, the packet is removed from the window, freeing a slot for a new packet. If the timeout of a sent packet expires and no Ack has been received, the packet is retransmitted. If the transmission window is full, new packets to send are stored in a buffer until a free slot appears in the transmission window. In the reception node, received packets are placed in order by their sequence number in a reception window, while their respective Acks are sent back to the transmitter. Would there be a missing packet in the reception window, a "not acknowledged" message (Nak) is sent back to the transmitter. Fig. 4 shows the implemented FSM diagram.

For this implementation, the following considerations were made:

- Packets to be transmitted turn up in sequence number order; reception of these packets in their correct order is ensured by the protocol.
- There is one transmitter and one receiver at each node, and communication is limited to two nodes.

The assignment of timeouts to packets were implemented as dictionaries. The event types used were Timeout, Ack, Nak or Data. Simulation tests were carried out using a `Virtual Channel` block. `Data source` and `Event Sink` blocks were used at the ends. Real wireless tests were carried out using the same physical layer as in in Fig. 3. This is basically a discrete digital communication system using QPSK at 850 MHz, including CRC checking, with gain, frequency, phase, and time corrections in reception. For this experiment, two USRPs B100 were used. Both simulated tests and real communication using ARQ proved successful as all data sent was received in the correct order.

Besides the specific implementation of the protocol, this instance proved the GWN framework to be within easy reach of new users, who successfully added new functionalities by creating new blocks, in a seamless extension to GWN.

9.2. CSMA/CA

The Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) is a data communications protocol used as a part of the Medium Access Control sublayer (MAC). As a part of a graduation project in telecommunications engineering, a simple CSMA/CA protocol was implemented using GWN.

The proposal was to implement CSMA/CA over a Stop And Wait ARQ protocol. Before sending a packet, the channel is

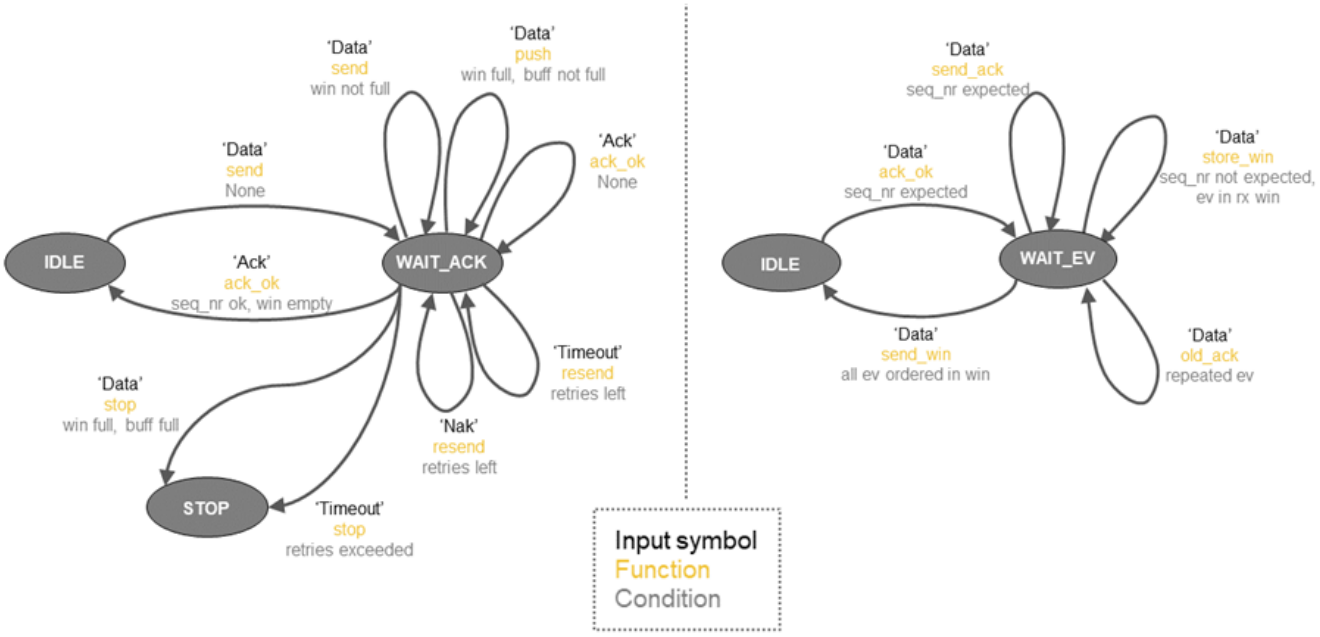

Figure 4: FSM implementations of transmitter (left) and receiver (right) for selective repeat protocol.

measured for activity. If the medium is busy, a exponential back-off algorithm starts a (random) timer. If the medium is free, the packet is sent. When the channel is busy and a timeout arrives, a new measure of activity in the medium is made. In case of activity, the process is restarted (random timer, with times possibly incremented). There is no RTS (Request To Send) nor CTS (Clear To Send) exchange in this simple version of a CSMA/CA, and the times of real wireless communications are not considered, as the goal of the project was not to fulfill the many requirements and efficiency of established protocols (such as IEEE 802.11) but to build a simple MAC with shared medium access control using the GWN and GNU Radio toolkit.

Activity in the medium is determined by a measurement of signal power at the USRP source block, by means of GNU Radio block `Probe Avg Mag^2`. The CSMA block included an FSM, importing the value of the Probe block as a condition to be read at transitions. The import was done through the usage of cheetah embedded in the XML code of the CSMA block, leading to a pretty straight way to read data into the CSMA's code.

The first test was a simulation with a self-generated noisy signal. The protocol was tested in real wireless communication by carrying out a three-people chat between USRPs. This implementation of the CSMA protocol was then used to communicate three autonomous mobile robots in a graduation project; to this purpose, it can optionally send debugging information to the chat client for measuring QoS (Quality of Service). Modulation was the same shown in 3 although adjustments had to be made for different USRP models, distances, and personal computers. Tests used combinations of USRP B100, B200, B200mini, and different kinds of personal computers.

This experience involved mainly by two undergraduate students, one of them with some prior work related to GWN, and the other without any experience in GWN / GNU Radio programming. The time and effort dedicated to the project was quite reasonable in view of the results obtained. This can be traced to the modularity and extensibility of design, as well as to the care given to documentation and tutorial material in GWN.

10. RESULTS AND CONCLUSIONS

The initial motivation for this project sprang from two different areas: research and education. While addressing questions on wireless data networks, such as cognitive networks and protocol efficiency, we felt the need of experimenting some ideas in real communication. In education, we carry on several courses on data networks and wireless communications, and were in search of a hands-on way for the students to appreciate the internals of data networking protocols, experiment their pros and cons, and attempt some improvements on their own ideas.

GNU Radio and the GNU Radio Companion seemed the ideal tool to face both needs, for several reasons: separation of functions in specific blocks, a graphical presentation, a practica-

ble extension mechanism, simulation, open and free availability, low cost wireless hardware, and accessibility on a low budget. Hence, the question became: how can GNU Radio be extended so as to allow real world communications demonstration and experimentation, while at the same time follow the design patterns of data networks protocols as closely as possible? The new framework should be friendly to data network specialists, or at least close to their expectations. After trying several approaches, the notion of an Event object for inter block communication, their conversion to and from GNU Radio PMTs, and the integration of Event inputs and outputs in a generic GWN block, proved to bridge the gap between wireless communication and data networks quite successfully. Handling of time and FSM machines were added and made accessible through the GWN block, thus completing the picture.

After some years of development, the GWN framework is now considered to be mature enough to be used both in research and education. It has been used in several editions of data network courses, both for demonstration and experimentation, and some undergraduate and graduate projects have used it for validation and measurement. Though the set of data network oriented blocks provided by GWN is still modest, it has proved to be enough for simple projects, and easily enhanced to provide new functions. Extensions are expected to come in a natural way as the use of GWN becomes more extended. To date, its main limitation is performance. Though this is a drawback of all SDR based projects, GWN blocks are nowadays coded in Python; rewriting `gwnblock` and some other essential GWN blocks in C++ is expected to improve on performance, even if other new blocks are coded in Python, a must for fast prototyping and wide accessibility.

In our experience, GWN has proved a valuable tool in education and research, and its use in industrial prototyping is not to be discarded. It has proved to be accessible to students and to network specialists as well, and extensions through the creation of new GWN blocks came to be quite straightforward. All these tools are open and free; they can not only be obtained immediately and at no cost, but can also be explored in its internals, modified, extended, or applied to new developments with no limitations. This opened a universe of potential realizations never seen before, and within reach of even very small budgets. This is very good news for underdeveloped countries.

REFERENCES

[1] Range Networks, "OpenBTS." [Online]. Available: http://openbts.org/

[2] Ettus Research, "How to build an FM receiver with the USRP in less than 10 minutes." [Online]. Available: https://kb.ettus.com/Implementation_of_a_Simple_FM_Receiver_in_GNU_Radio

[3] F. Larroca, P. Flores Guridi, G. Gómez Sena, V. González Barbone, and P. Belzarena, "An open and free ISDB-T full_seg receiver implemented in GNU Radio," in *Wireless Innovation Forum Conference on Wireless Communications Technologies and Software Defined Radio (WInnComm '16)*, 2016.

[4] GNU Radio, "GNU Radio webpage." [Online]. Available: https://www.gnuradio.org/

[5] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless mac processors: Programming mac protocols on commodity hardware," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1269–1277.

[6] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "An IEEE 802.11a/g/p OFDM receiver for GNU Radio," in *Proceedings of the Second Workshop on Software Radio Implementation Forum*, ser. SRIF '13. New York, NY, USA: ACM, 2013, pp. 9–16.

[7] ARTES, "GWN, the GNU Radio Wireless Network project, homepage." [Online]. Available: https://github.com/vagonbar/gr-gwn/

[8] N. Spurrier, "Noah Spurrier's FSM." [Online]. Available: http://www.noah.org/python/FSM

[9] ARTES, "GWN, the GNU Radio Wireless Network project wiki." [Online]. Available: https://github.com/vagonbar/gr-gwn/wiki