Supporting Integrated MAC and PHY Software Development for the USRP SDR

Rahul Dhar, Gesly George, Amit Malani
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15215

Peter Steenkiste
Computer Science and Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15215

Abstract— **Software Defined Radios (SDR) offer great runtime flexibility both at the physical and MAC layer. This makes them an attractive platform for the development of cognitive radios that can adapt to changes in channel conditions, traffic load, and user requirements. However, to realize this goal, we need a software framework that supports both MAC protocol and PHY layer development in an integrated fashion. In this paper we report on our experience in using two different software frameworks for integrated PHY-MAC development for SDRs: GNU Radio, which was originally designed to support PHY layer development, and Click, a framework for protocol development. We also discuss a number of broader system considerations, such as what functionality should be offloaded to the SDR device.**

Keywords - software-defined radio; sofware framework; USRP; GNU radio; Click

I. Introduction

Software Defined Radios (SDR) offer great flexibility for runtime adaptation to the signal environment (e.g. spectrum availability, interference, ..), so they can support cognitive radios that automatically adapt to the environment. Moreover, if the flexibility at the radio level can be coupled with adaptive MAC protocols, SDR platforms open the door for runtime cross-layer optimizations. In combination, the MAC and PHY layers can adapt not only to the signal propagation environment, but also to application and user requirements. Such a "cognitive radio" can greatly improve the efficiency of spectrum use and observed user network performance.

To realize this goal, we need a software framework that supports both MAC protocol and PHY layer development in an integrated fashion. However, PHY and MAC layers are often developed by different communities using different tools so no such integrated platform exists. Several software environments have been developed for implementing PHY layers for SDR platforms. Besides a number of commercial platforms, two well known open source platforms are GNU Radio and the Virginia Tech OSSIE platform. MAC layer development for SDR has received much less attention, but several groups have developed general-purpose protocol frameworks [4][22][23]. Some of these can be used to implement wireless MAC protocols, e.g. [24][25].

In this paper we report on our experience in using two different software frameworks for the integrated development of MAC and PHY layers for SDRs. Out target platform is the Universal Software Radio Peripheral (USRP) device built by Ettus Research [1] and we focus on the use of SDRs for data networking. The first framework we used is the GNU Radio framework, which was originally designed to support PHY layer development. We found that while we were able to integrate a basic MAC layer into GNU Radio, some key MAC layer functionality was missing and would have to be added. Second, we explored the use of Click, a framework that was specifically designed to support the development of communication protocols. We ported Click to the USRP and, since Click does not support wireless PHY layer functions, we also developed a mechanism that allows us to port GNU Radio modules to Click in a systematic manner. We describe our "Click with PHY" implementation and compare its design with other design options. Finally, during our work on implementing MAC protocols for the USRP, we identified a number features that could not easily be implemented fully in software and can benefit from hardware support on the USRP device.

This research was funded in part by NSF under award No. CNS-0520192.

The remainder of this paper is organized as follows. In the next section, we introduce the USRP device and the GNU Radio framework. In Section 3, we report our experience in using GNU Radio for MAC protocol development. In Sections 4 and 5, we introduce Click and compare a number of possible designs for using it to support integrated PHY-MAC software development. We describe a specific Click-based PHY-MAC framework implementation in Section 6 and we discuss broader systems considerations in Section 7. Finally, we present related work in Section 8 and we summarize in Section 9.

II. USRP and GNU Radio

In this section, we briefly introduce the USRP device and the internal organization of GNU radio.

A. USRP

The Universal Software Radio Peripheral (USRP) is a basic SDR platform [20]: it implements front-end functionality and A/D and D/A conversion, but it is assumes that physical layer processing will be done on the PC that hosts the device. The USRP connects to the PC using the Universal Serial Bus (USB2). The typical I/O stream is 32 bits of I/Q samples - 16 bits each for both the in-phase and quadrature component. Since the maximum USB2 rate is 60 MB/sec, the USRP can theoretically transfer 15 Msamples/sec, yielding a maximal spectral bandwidth of 7.5 MHz. Some hosts cannot achieve this rate because they have a slower USB implementation.

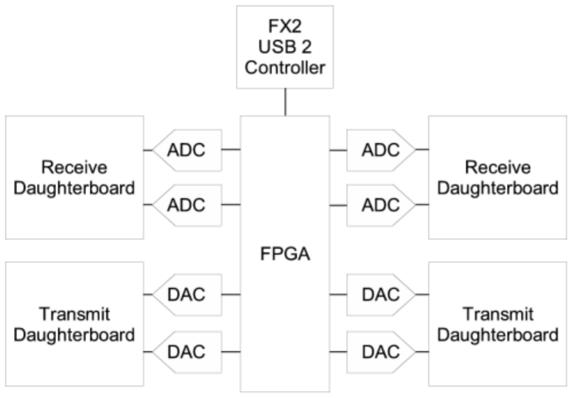

Figure 1. Block Diagram of the USRP (from [18])

The USRP device (Figure 1) consists of a motherboard containing up to four high speed 12-bit 64M samples/sec Analog to Digital Converters (ADC), four high speed 14-bit 64M samples/sec Digital to Analog Converters (DAC), an Altera FPGA and a programmable Cypress FX2 USB 2.0 controller. The ADCs, DACs and the FPGA together provide support for IF processing. The FPGA on the board provides four Digital Up Converters (DUC) and four Digital Down Converters (DDC) to shift frequencies from the baseband to the required frequency. This means that the RF regions handled by the daughter cards can be split into one, two, or four channels. The FPGA can be reprogrammed to provide additional functionality. The USRP provides data buffers in both the FX2 and the FPGA. Both components maintain separate buffers for the TX and RX paths. The FX2 provides 2 KB each for TX and RX, and the FPGA provides an additional 4 KB each.

RF front ends are attached in the form of daughter cards. Various daughter cards are available on the Ettus web site. The cards that are most relevant for data networking include the basic, Flex400, and Flex2400 cards; they operate at coax, 400-500 MHz, and 2300-2700 MHz frequencies, respectively. The research presented in this paper uses the basic card since it was the only card that was available when the research started.

The USRP uses the GNU Radio framework for PHY layer processing on the PC. We describe GNU Radio next.

B. GNU Radio

GNU Radio is an open source toolkit for building software radios [9]. It is designed to run on desktop computers and, combined with minimal hardware, allows the construction of simple software radios. The project was started in early 2000 by Eric Blossom and has evolved into a mature software infrastructure that is used by a large community of developers.

The GNU Radio signal processing library provides signal processing blocks for modulation, demodulation, filtering, and I/O operations such as file access. In addition, it also provides blocks for communicating with the USRP. New blocks can be added as needed. A radio is built by connecting these blocks to form a *flowgraph*. This flowgraph is a directed acyclic graph in which the vertices are the *GNU Radio blocks* and the edges correspond to data *streams*. Figure 2 shows how a FIR Filter, Quadrature Demodulator and Audio Sink are connected in a flowgraph to form a simple FM receiver. Programming in the GNU Radio platform uses a combination of C++ and Python: the processing blocks are implemented in C++ while the flowgraph and the applications that sit on top are developed in Python. We now briefly elaborate on key properties of both processing blocks and flowgraphs.

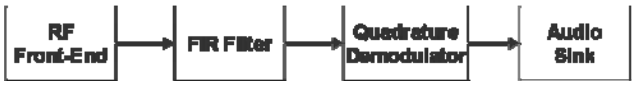

Figure 2. GNU Radio flowgraph for a simple FM Receiver

Processing blocks - Generally blocks operate on continuous streams of data. Most blocks have a set of input and output streams: they consume data from their input streams to generate data for their output streams. Special blocks, called *sources* and *sinks*, only produce or consume data, respectively. Examples of sources are blocks that read from USRP RX ports, sockets and file descriptors. Similarly, sinks include blocks that write to USRP TX ports, sockets and file descriptors. Each block has an input and output signature (IO signatures) that defines the minimum and maximum number of input and output streams it can have, as well as the size of the data type on the input and output streams.

Each block defines a work function that operates on its input to produce output streams. In order to help the scheduler decide when to call the work function, blocks also provide forecast functions that tell the runtime system the number of input items it requires to produce a number of output items and how many output items it can produce given a number of input items. At runtime, blocks tell the system how many input (output) items they consumed (produced). Blocks may consume data on each input stream at a different rate, but all output streams must produce data at the same rate.

Data buffers – The input and output streams of a block have buffers associated with them. Each input stream has a read buffer, from which the block reads data for processing. Similarly, after processing, blocks write data to the appropriate write buffers of its output streams. The data buffers are used to implement the edges in the flowgraph: the input buffers for a block are the output buffers of the upstream block in the flowgraph. GNU Radio buffers are single writer, multiple reader FIFOs.

Flowgraph mechanisms – Users build a radio by defining a flowgraph using the `connect` function. The `connect` function specifies how the output stream(s) of a processing block connects to the input stream of one or more downstream blocks. The flowgraph mechanism then automatically builds the flowgraph; the details of this process are hidden from the user. An key function during flow graph construction is the allocation of data buffers to connect neighboring blocks. The buffer allocation algorithm considers the input and output block sizes used by blocks and the relative rate at which blocks consume and produce items on their input and output streams. Once buffers have been allocated, they are connected with the input and output streams of the appropriate blocks.

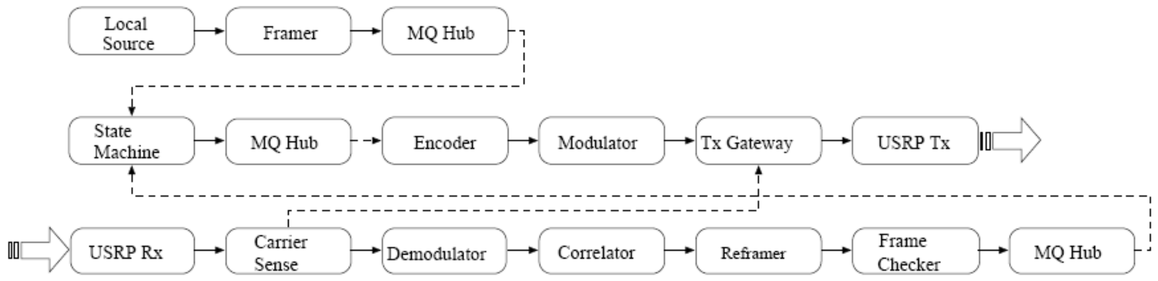

Figure 3. Flowgraph for MAC protocol

Scheduler - The GNU Radio scheduler executes the graph that was built by the flowgraph mechanism. It is implemented as a single thread that loops over all the blocks in the graph, executing each block sequentially until all the data has been consumed. During the execution, the scheduler queries each block for its input requirements and it uses the above-mentioned forecast functions to determine how much data the block can consume from its available input. If sufficient data is available in the input buffers, the schedule calls the block's work function. If a block does not have sufficient input, the scheduler simply moves on to the next block in the graph. Skipped blocks will be executed later, when more input data is available. The scheduler is designed to operate on continuous data streams.

III. SUPPORTING MAC DEVELOPMENT

In this section we report our experience in adding a CSMA/CA MAC protocol to GNU Radio. This research was done in Spring 2005, using GNU Radio version 2.5.

A. MAC development in GNU Radio

The MAC protocol is often specified as a state machine that defines what actions must be taken in response to specific events. Our state machine (Figure 4) represents a very simple protocol that handles only one frame at a time and rejects all frames except the one that it expects. It does not implement timeouts since they are not supported by GNU Radio. It has four states: IDLE, WAIT FOR CTS, WAIT FOR DATA, and WAIT FOR DATA-ACK. The state machine starts in the IDLE state, where it waits for a frame to arrive via a message queue (MQ). A frame can either be a locally generated frame or an RTS from a remote node. The state machine handles incoming frames before locally generated frames. The WAIT FOR DATA-ACK state is interesting because it can transition to one of three states after processing a DATA-ACK. If there is an RTS in the MQ for local frames, it transmits it and enters the WAIT FOR CTS state. If there is a pending RTS from another node, it emits a CTS and enters the WAIT FOR DATA state. If there are no frames in either MQ, it returns to the IDLE state.

Flowgraph – The simplest way to realize the MAC protocol is as a single block that implements the state machine. This block would combine the transmit and receive data paths and it would have separate input channels for locally generated and incoming frames. This design is however not possible because of constraints placed on GNU Radio flowgraphs. For example, all input channels must have the same data rate.

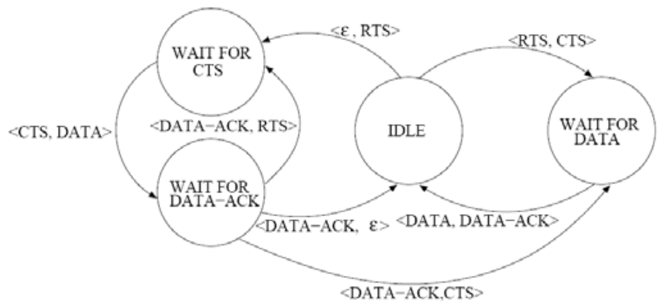

Figure 4. State diagram for simple CSMA/CA MAC protocol

The alternative is to implement separate pipelines for transmit and receive, as is shown in Figure 3. Not only do the transmit and receive data paths execute in their own threads, but the state machine and frame emission process also runs in a separate thread. This is necessary to allow the state machine to block (e.g., on I/O or on a pseudo-timer) without halting any frames that have already been marked for transmission. In order to make sure these three functions can execute independently, we use a Message Queue (MQ) Hub. This block takes in data and enqueues it in a message queue, as shown by the dashed lines. It acts as a sink, causing any path that uses it as a terminal to be executed as a thread.

Framing - We use a very simple frame format. The header includes the following fields: synchronization field, source and destination address, sequence number, frame type, payload length, and checksum. This is followed by the payload and optional padding. Because of restrictions in the GNU Radio blocks we used for coding, all packets are 128 bytes long. The first field is a synchronization code with good autocorrelation properties that allows the receiver to lock on to the packet.

Data to be transmitted is passed to GNU Radio via a file descriptor. The Local Source block breaks the data stream into blocks of MaxPayloadSize/2 bytes. For each pair of blocks it creates two packets: an RTS and a data packet of length MaxPayloadSize. Data received from the network is converted to a byte stream by the correlator and converted into frames by the reframer block. The reframer drops the synchronization field and passes the block to the frame checker block for validation. Validation entails verifying the checksum, ensuring the frame type is valid, and checking the destination address. Validated frames are enqueued in the MQ Hub, where they are read by the state machine. For coding and modulation we used existing GNU Radio blocks: the NRZ block for coding and the GMSK or FSK blocks for modulation [2].

Carrier Sense - The Carrier Sense block receives the sampled signal from the USRP RX port and calculates its power. If the power is greater than the carrier sense threshold `thresh`, it signals that a carrier is present. If the power drops below `thresh`, it signals that there is no carrier. Signaling is performed by sending a message to the Tx Gateway block, which keep track of whether a carrier is present of not. It only allows data to be transmitted if no carrier is present. Otherwise, it delays transmission until it receives the NOCARRIER message from the carrier sense block.

It turned out to be difficult to use Carrier Sense because when using the basic daughter cards, the hardware continues to transmit between packets. An alternative is to use virtual carrier sense: the receiver continuously tries to decode frames and it reports that the channel is idle if no frame is detected.

Testing – Running the MAC protocol over the USRP using the basic cards turned out to be challenging. As discussed in more detail below, the USRP basic cards we used were designed to operate in stream mode and it was difficult to use them in packet mode in a reliable manner. Moreover, the lack of timers made it difficult to recover from errors. For this reason, we mostly debugged and tested the MAC protocol without the hardware. We ran two versions of GNU Radio, representing two nodes, on the same PC and connected their transmit and receive blocks back-to-back using named pipes.

B. *Lessons learned*

USRP and GNU Radio proved to be excellent platforms for experimenting with software radio. Even though we used both platforms in ways that clearly fell outside the scope of their original design, we were able to complete a basic MAC-PHY protocol. However, we did find that some MAC protocol features were difficult or impossible to implement in GNU Radio. We give an overview in this section; more details can be found in [2].

The GNU Radio framework is well suited for implementing independent transmit or receive data paths, e.g. an FM receiver, but MAC protocols often need to transmit and receive in a coordinated fashion. Unfortunately, the GNU Radio scheduler executes the flow graph sequentially from a source to a sink, so combining transmit and receive functions in a single flowgraph is difficult. While it is possible to combine flowgraphs, as we described above, there is no mechanism to explicitly coordinate them, e.g. to force a transmit immediately after a receive.

GNU Radio was designed to support signal processing on continuous data streams. There is no concept of (fixed or variable sized) packets or frames. The stream-centric design is most prominent in the flowgraph mechanisms, specifically buffer management and scheduling. Flowgraphs are a direct realization of radio block diagrams, so they are a perfect fit for radio design. One of their benefits is that buffer sizes can be left unspecified and can be automatically derived based on the data units and relative rates of the input and output streams of blocks. Unfortunately, this does not work well for frames, since the relationship between frames and groups of bytes or signal samples is highly variable. For example, a block may not know a priori how many bytes it needs to consume on its input stream to generate a frame on its output stream; it may need to interpret the packet header first.

Similarly, flowgraphs support the automatic scheduling of processing blocks based on input and output properties of blocks. MAC protocols have however more complex scheduling requirements that cannot be automatically derived. Also, MAC protocols may want control over the scheduling of different actions (e.g. sending versus receiving). Finally, at specific points in the stack, a MAC protocol may want to processes frames in non-FIFO order (e.g. based on priority). Unfortunately, GNU buffers only support FIFO access, which is sufficient for signal processing.

Since GNU Radio was designed for signal processing, it does not provide support for maintaining global state. MAC protocols generally maintain a state machine, which is updated by both the transmit and receive paths and is used to coordinate access to the shared medium. Moreover, MAC protocols often need to keep per-flow or per-destination state, e.g. transmission parameters, flow control information, bandwidth use, etc. Finally, the GNU Radio framework lacks the concepts of time and timers. MAC protocols need support for timers, for example, to implement back off mechanisms, various inter-frame gaps, or TDMA-style gaps.

C. *Discussion*

Given that GNU Radio was designed for PHY layer processing, and given the big differences in the requirements for the PHY and MAC layers, it should not be a surprise that GNU Radio does not support MAC protocol development "out of the box". This leaves us with two alternatives to support integrated PHY and MAC layers processing: either we can extend GNU Radio with MAC layer support, or we leverage an existing protocol framework for MAC layer support. The first option is being explored by a group at BBN (see Section VIII). We explore the second option in the remainder of this paper.

IV. CLICK OVERVIEW

Click is an open-source, modular software architecture for building reconfigurable routers [4]. We selected Click because it has been fairly widely adopted and has been successfully used to implement a variety of network protocols. In this section we give an overview of Click's internal structure.

A. *Click overview*

Click routers are built from fine-grained components, called *elements*, that perform packet processing [5]. A protocol is built as a directed graph of elements. The graph's edges, called *connections*, represent possible paths for packet handoff between elements. Elements can have any number of input and output *ports* that are used to connect elements together, as described below. Elements have an optional configuration string that can be used to specify parameters during initialization. The Click distribution provides a large number of elements implementing common routing functions, e.g. device handling, routing table lookup, queuing, etc. Click is implemented in C++ and it can be run as a kernel module as well as a user-level process.

Click supports two types of connections: *push* and *pull*. They implement complementary forms of packet transfer. Both are implemented as procedure calls. On a push connection, the source initiates a packet transfer downstream to the destination element. In contrast, on a pull connection, the packet transfer is initiated by the destination element. Through a series of upstream packet transfer requests, it asks the source element to return a packet, if available. Figure 5 shows a Click router configuration that has both push and pull connections. Packets are transferred between adjacent elements as part of the *push(p)* calls and as part of the *return* from the *pull()* calls.

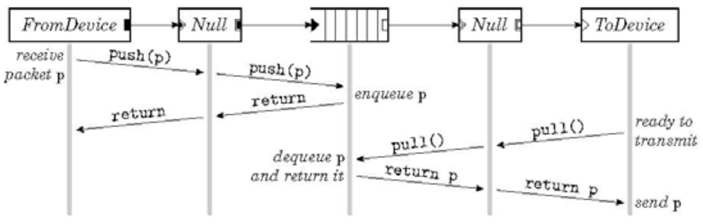

Figure 5. Push and Pull Connections (from [4])

Connections between elements are determined by the types of the ports at its endpoints. All ports of an element are either *pull* ports or *push* ports. A pull connection sits between two push ports while a pull connection connects two pull ports. One cannot have a connection between a push port and a pull port. Elements can also have *agnostic* ports, which behave as push or pull ports, depending on what port they are connected to.

Click elements do not have implicit queues on their input or output ports. Queues that store packets are implemented by a separate *Queue* element. This gives the developer the flexibility to decide where and how packets should be stored in the router. A *Queue* element has a push input port and a pull output port; the push input port enqueues pushed packets and the output port dequeues pulled packets and returns them. The middle element in Figure 5 is a Queue element.

Click supports both explicit and implicit scheduling. Elements that require special access to the CPU are explicitly scheduling using either *Tasks* or *Timers*. Elements that require frequent access to the CPU are defined as tasks and are placed on a task queue. The Click router processes the task queue in a loop one element at a time. The task queue is scheduled with the flexible and light weight stride scheduling algorithm [5]. Any element that frequently initiates push or pull requests without receiving a corresponding request should be placed on the task queue. In Figure 5, the *FromDevice* and *ToDevice* are scheduled as tasks since they need to be executed frequently. Elements that should execute at a specific time can be scheduled using timers. An element can have any number of active timers. When a timer fires, it executes an arbitrary function defined by the user. Timers are checked relatively infrequently [6], so there could be a considerable delay between a timer's nominal expiration time and the actual time it runs. Since Click uses cooperative scheduling, timer callbacks should run for only a short period of time.

Most elements in Click are implicitly scheduled by the push/pull connections between elements. When an element placed on the task queue is processed, it initiates a sequence of either push or pull requests that invoke each element in the graph. All the elements in Figure 5, other than *FromDevice* and *ToDevice*, are implicitly scheduled using push and pull.

Table I summarizes the differences and similarities between Click and GNU Radio.

TABLE I. COMPARISON BETWEEN GNU RADIO AND CLICK

GNU Radio	**Click**
Executes a directed graph	Executes a directed graph
Blocks process data streams	Elements process packets
Blocks are written in C++	Elements are written in C++
Buffer management is implicit – managed by the flow graph mechanism	Buffer management is explicit - managed by a Queue element
Each block is explicitly scheduled by the scheduler; there is no support for timers	Most elements are scheduled using push/pull; elements can also be triggered by timers
The flowgraph mechanism and scheduler together manage the internal buffers	Buffer management is done by the queue element.

V. CLICK AND GNU RADIO INTEGRATION ALTERNATIVES

We describe three different ways of combining GNU Radio and Click and compare them with respect to development cost. More specifically, there are three types of development activity we need to consider:

- Development of a protocol stack with both MAC and PHY layer blocks in Click for use with the USRP
- Porting GNU Radio blocks to Click
- Development of the integrated Click/GNU framework and its maintenance as new release of Click and GNU radio become available.

Our priority is to optimize the first type of development, i.e. it should be relatively easy to build integrated PHY-MAC protocol stacks for USRP. In this context, "easy" means that it requires a small number of lines of code and only a minimal understanding of the framework to use GNU Radio blocks. Initial porting effort has the lowest priority.

We now discuss the three design alternatives in more detail. Two of these integrate GNU Radio PHY blocks into Click, either by encapsulating a GNU Radio flowgraph in a single Click element or by encapsulating individual GNU Radio blocks in Click elements. The third option uses Inter Process Communication (IPC) to connect Click and GNU Radio processes. Since the first two options require Click to use the USRP device, we discuss this task first.

A. *Interfacing USRP with Click*

The first two design choices require Click to interface with the USRP for transmitting and receiving packets. This requires two new Click elements, *ToUsrpDevice* and *FromUsrpDevice*, to replace the standard elements *FromDevice* and *ToDevice* that Click uses to communicate over the network. These elements can be developed in two ways. First, we can implement custom elements, using the USRP library to read from and write to the USRP. Alternatively, we can port the corresponding GNU Radio blocks to Click.

We chose the second option because these blocks, called *usrp_source* and *usrp_sink*, are similar to other GNU Radio blocks, so we can leverage our infrastructure for porting GNU Radio processing blocks. Specifically, the *FromUsrpDevice* and *ToUsrpDevice* elements must allocate the buffers needed for the GNU blocks and must then explicitly call the work function of the blocks. The *FromUsrpDevice* is a push element that pushes the data read from the USRP buffer to the next Click element in the directed graph. The *ToUsrpDevice* is a pull element that pulls data from the previous element in the directed graph and writes it to the USRP buffer. Both these elements are placed on Click's task queue. They implicitly schedule the other elements in the directed graph.

B. GNU Radio Flow Graph as a single Click Element

We encapsulate a directed graph of GNU Radio blocks as a single Click element and the graph of blocks is executed when the Click element containing it is executed by Click. This approach requires porting GNU Radio's flowgraph mechanism and scheduler (Section II.B) to Click. This can be most easily done by using the Click *helper class* mechanism, which supports the implementation of non-element Click classes. In GNU Radio, the flowgraph mechanism, which includes the buffer allocation process, is implemented in Python; this needs to be ported to C++. The GNU Radio scheduler, which calls the work functions of the blocks in the flow graph, is already implemented in C++. The scheduler must be invoked for each execution of the Click element. The source in the flow graph must obtain its data from the previous Click element, while the sink should pass the processed data to the next Click element.

Once the flowgraph mechanism and scheduler are available in Click as helper classes, creating a Click element that encapsulates GNU Radio blocks is fairly simple. A first step is to pass the required blocks to the flowgraph mechanism during the initialization phase of the Click element so it can create the flowgraph; the required blocks can be identified from the existing Python scripts used in GNU Radio. The next step is to execute the scheduler on the flowgraph whenever a push/pull request is executed on the element. Figure 6 shows the directed graph in Click for an element that does GMSK modulation.

The effort involved in using this approach for the different forms of development can be summarized as follows:

- One-time effort to port the flowgraph and scheduler mechanisms and to port the *FromUsrpDevice* and *ToUsrpDevice* elements. New releases of GNU Radio or Click may require changes to the ported modules.
- Protocol stack developers must define their PHY layer as a GNU Radio flowgraph, which will be realized as a Click element. This requires writing code that passes the necessary blocks to the flowgraph mechanism. This approach does give the developer the flexibility to do cross layer optimizations.

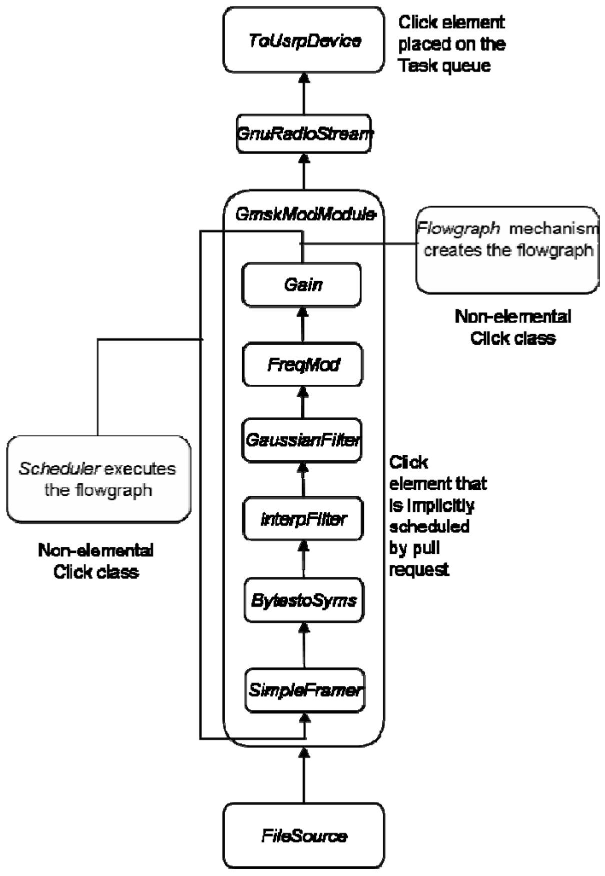
ToUsrpDevice
Click element placed on the Task queue
GnuRadioStream
GmskModModule
Flowgraph mechanism creates the flowgraph
Non-elemental Click class
Gain
FreqMod
GaussianFilter
InterpFilter
BytestoSyms
SimpleFramer
Click element that is implicitly scheduled by pull request
Scheduler executes the flowgraph
Non-elemental Click class
FileSource

Figure 6. GMSK Modulation in Click using Approach One

C. Each GNU Radio block as a separate Click Element

An alternative design is to encapsulate each GNU Radio block as an separate Click element. In this approach, the flowgraph of blocks is represented as a directed graph of Click elements, and the GNU Radio block inside each Click element is scheduled when the element is executed by Click. For this to work, each Click element will have to handle buffer allocation, scheduling, and the exchange of data with other elements.

In GNU Radio, buffer allocation is handled by the flowgraph mechanism based on the input and output properties of the elements. In Click, each element will not only need to know the properties of its own block, but also those of the block encapsulated in the downstream element. This can be done by having elements pass buffer size information or pointers to the block. To execute the encapsulated block during a push/pull request, each element will need to explicitly call the work function of the block and update the buffer pointers. For this, the element will need to call the block's forecast functions to determine how much input the block can currently consume. Finally, push/pull requests transfer packets while GNU Radio blocks operate on streams. To deal with this mismatch, a stream class must be created so that the pointer to the output buffer of a block can be passed to the block encapsulated in the next element as its input buffer. The *GnuRadioStream* block in Figure 7 represents this stream class. This process must be repeated for every GNU Radio block that is ported.

For the three types of development, the effort involved can be summarized as follows:

- One-time effort to port the flowgraph and scheduler mechanisms and to build the *FromUsrpDevice* and *ToUsrpDevice* elements.
- Porting a new block involves writing a new Click element that encapsulates the block, allocating the buffer for the block, calling its work function and using the Click-USRP element as part of the flowgraph. This approach requires the developer to understand the GNU Radio scheduling and flowgraph mechanisms. Changes to Click or GNU Radio may require modifications to all ported blocks.
- A stack developer can use a directed graph of Click elements that encapsulate blocks along with "regular" Click element to implement PHY and MAC layer functionality. This gives the developer the opportunity to carry out cross layer optimizations.

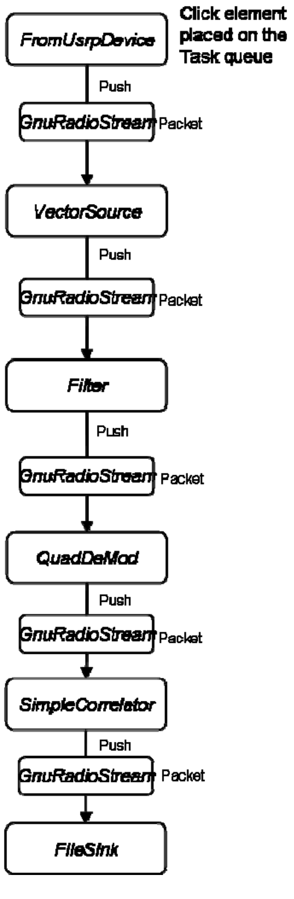

Figure 7. Demodulation in Click using Approach Two

D. IPC between Click and GNU Radio processes

Click and GNU Radio execute as individual processes with Click implementing a MAC protocol and GNU Radio performing PHY processing. The two processes communicate via pipes or message queues. This approach allows Click and GNU Radio to execute with minimal changes and porting effort. The Click element that communicates with GNU Radio as well as the python script that is run in the GNU Radio environment will need to setup the IPC primitives.

For the different types of development, this approach involves the following costs:

- One time effort to develop support in Click and GNU Radio to set up IPC.
- There is no porting effort for using PHY blocks.
- For a stack developer, this approach does not offer easy integration opportunities between MAC and PHY. There is effort involved in understanding and using appropriate IPC mechanisms to interface between Click and GNU Radio processes.

E. Discussion

Based on the above design analysis, we decided to implement the first approach, i.e. to embed a flowgraph of GNU Radio blocks into a single Click element. The advantage of this approach is that, once the scheduler and flowgraph mechanism have been ported to Click, writing a Click element that encapsulates a flowgraph of blocks is a simple, mechanical process. The first step is instantiating objects for the blocks and passing a vector of these object pointers to the flowgraph's connect function in the element's initialization function. The second step consists of a call to the flowgraph's start function to initialize the scheduler. The final step involves executing the scheduler on the blocks by calling the flowgraph's run function when the element is executed.

For the example in Figure 6, this requires the developer to write six lines of code to create the six blocks, six lines to add them to a vector, three lines to initialize and pass the vector to the flowgraph, and one line to call the flowgraph's run function to execute the scheduler – a total of sixteen lines of code. The functional behavior of the flowgraph and the scheduler is completely hidden from the developer.

In contrast, using and porting GNU Radio blocks in Click is much more cumbersome with the second approach. For each block, one needs to explicitly allocate the buffer and call the work function of the block to execute it. This requires knowledge of the buffer sizes of this block as well as the type of the block used in the downstream Click element to ensure appropriate buffer sizes. One also needs to understand the stream class to pass the processed data to the next element. This approach not only requires more coding, but also a deeper understanding of buffer management and block architecture.

The first two approaches require similar effort when GNU Radio is changed, e.g. when the scheduler is modified. In the first approach, these modifications will need to be incorporated into the ported scheduler helper class in Click, while in the second approach, similar changes need to be made to the GNU Radio Blocks used in Click. The complexity of this process depends on the extent of modification made to GNU Radio.

With the third IPC-based approach, initial integration and dealing with new releases requires minimal effort: it only requires an understanding of IPC mechanisms. It does however have several disadvantages: developers of protocols need to become familiar with two frameworks (Click and GNU Radio), it is more difficult to implement cross-PHY-MAC layer interactions, and it is potentially less efficient since it requires inter-process communication.

VI. IMPLEMENTATION

We give a brief overview of our integrated Click-GNU Radio prototype and describe how it was used to implement a simple TDMA protocol.

A. *Implementation overview*

We implemented the integrated framework as described in Section V.B. Our implementation is based on Click version 1.4.3 and GNU Radio version 2.8.

We tested the implementation with a 2-node network consisting of two USRP boards connected to two PCs running Linux. We used the basic daughter TX and RX cards operating at 29.32MHz for transmission and reception of signals. Initially, the TX and RX cards on the two USRP boards were connected together using SMA cables connected to SMA tees to form a shared medium. The data rate was set to 100 KSamples/sec, the TX interpolation rate was 80, and the RX decimation rate was 160. These parameters are typical for the TX and RX basic daughter cards.

Our test application performed a simple file transfer. We observed during test runs that the first frame was always transferred correctly while later frames were often corrupted. We discovered that this problem was caused by the basic TX cards. Since the cards were designed for stream-oriented communication, they continue to transmit, even after all the data in the USRP transmit queue has been sent [7]. This transmission interferes with later frame transmissions by other nodes, often resulting in the corruption of those frames. Similarly, we had problems with packet sizes larger than 64 bytes, since they require multiple transmissions. Note that other USRP daughter cards, which became available after this work was completed, do support packet-based communication.

When the above test setup was modified to send and receive frames along independent path i.e. by directly connecting a TX card to the RX card on the other node and vice versa, the frame transmissions were found to be perfect as the receiver was always able to properly decode frames.

B. *A simple TDMA MAC*

In order to test the functionality of our integrated framework, we implemented a simple Time Division Multiple Access (TDMA) protocol. The TDMA protocol divides the channel into time slots that are statically allocated to the nodes sharing the channel. Figure 8 shows the Click graph for the TDMA protocol. The graph of *FromUsrpDevice* to the *DataSink* is the receive path and the graph of *DataSource* to *ToUsrpDevice* represents the transmit path. The *ToUsrpDevice* element is scheduled using timers, so that it sends data to the USRP only in its time slot. At that time, it pulls a frame from the data source and transmits it. The *FromUsrpDevice* is continuously placed on the task queue, as the RX daughter board constantly listens for transmissions on the medium.

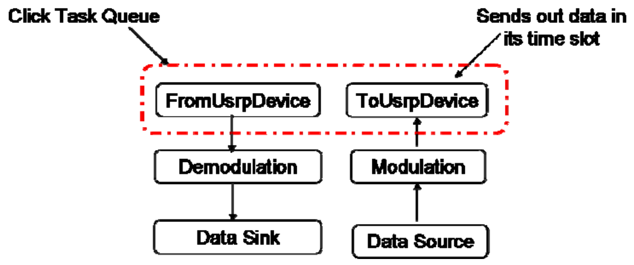

Figure 8. TDMA protocol using the GNU Radio with Click platform

The *Demodulation* and *Modulation* elements in Figure 8 represent the Click elements that encapsulate PHY processing. We used the blocks from the GMSK (Gaussian Minimum Shift Keying) implementation in GNU Radio. Figure 6 shows the GNU flowgraph that was encapsulated in the Click element. The demodulation element consists of a low pass filter, an FIR filter, an integrate filter and a correlator. We used a very simple frame format. The frame has a nine byte header that includes an eight byte synchronization code (preamble) and a one byte destination address, a 54 byte payload and a one byte trailer pad. The one byte tail pad is used to ensure that the correlator recognizes consecutive frames correctly.

We tested the TDMA MAC using the 2-node set up described earlier. The test consisted of the nodes exchanging a text file. The nodes transmitted frames in alternating time slots. On the receive path, each node dumps the payload of the frame that has its address as the destination into a text file. We found that the TDMA protocol worked well. We tried different lengths for the time slot and found that the time slots needed to be fairly long, e.g. a second or longer. The reason is that the packets were scheduled by the OS and that the clocks on the two nodes were not synchronized well. The time slots could clearly be reduced by improving clock synchronization and by adding fine grain timers (see Section VII). These tests offer preliminary evidence that Click, augmented with GNU Radio blocks, can support MAC and PHY layer processing for software radios. Clearly a more thorough evaluation is needed of both the system's functionality, by implementing richer MAC protocols, and its performance.

VII. DISCUSSION ON SYSTEMS ISSUES

We discuss the interactions between the integrated PHY-MAC framework and the rest of the system.

A. *Protocol stack and application interactions*

In a normal network stack, the MAC protocol runs in the kernel as part of a complete protocol stack. GNU Radio, however, is a regular user-space process, subject to the OS scheduler. On the typical computer, the user runs several traditional applications (e.g., web browsers and e-mail clients) with which the SDR software will have to compete. This is likely to result in less predictable performance.

We also need to consider how an integrated PHY-MAC layer implemented in GNU Radio can interact with the other protocol layers (IP and TCP/UDP, which typically run in the kernel). Figure 9 shows a possible configuration, using the TUN/TAP driver's TAP component which provides a virtual Ethernet net device [26]. This configuration involves several user-kernel crossing, which adds significant overhead. However, this configuration might be practical for low speed networks. An alternative is to move the entire protocol stack into user space, which would reduce the number of user-kernel crossings to one.

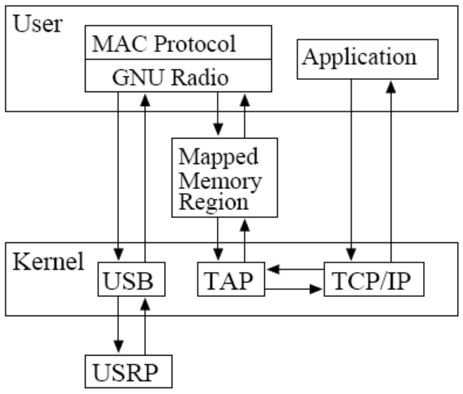

Figure 9. Possible integration of SDR software

Click can be executed in both user mode and kernel mode. Our integrated PHY/MAC framework was implemented in user-mode Click and it could use a configuration similar to Figure 9. However, by switching to kernel-mode Click, we could use a simpler configuration where PHY-MAC processing is done in the kernel, thus avoiding the extra user-kernel crossings. Note that this would result in a significant amount of in-kernel (PHY layer) processing, which may be undesirable, e.g. result in slow response times.

B. Device support for MAC protocols

Even with an appropriate software framework for MAC protocol implementation, there are a number of operations that are difficult or impossible to implement entirely in software on the host. The reason is that the USB is a relatively high latency, low bandwidth path that, combined with the general-purpose OS on the PC, results in unpredictable delays between the host and the network. We started to explore offloading three MAC support functions to the USRP [11]: timers, packet buffering, and carrier sense. These functions could for example be implemented on the FPGA on the USRP.

Timers: While it is possible to implement timers on the host (e.g. as is done in Click), they are relatively coarse grained and the USB adds an additional unpredictable delay. An alternative is to use the FPGA on the USRP to implement timers that can be used to control events, such as the start of a packet transmission. The timers could also be used to associate timing information with events, e.g. the end of (an assumed) packet reception.

Packet buffering: The USRP uses a single FIFO for each transmit card. The FIFO holds a stream of samples and does not recognize packet boundaries. Packet-based communication could benefit from being able to store several packets on the USRP. This might make it possible to pre-stage packets over the USB or to keep packets on the USRP for retransmission, thus reducing the USB load and delay.

Carrier Sense: Implementing carrier sense on the host introduces problematic delays. A first delay is associated with passing the samples from the USRP to the host over the USB. Moreover, when the host observes that the channel went idle, there is a delay associated with asking the USRP to start transmitting a packet. These delays can be avoided by implementing carrier sense on the USRP.

The above functions will make it possible to control timing more precisely and to reduce some delays. This will become more important as timing requirements become tighter, i.e. as transmission rates go up. Note that in an architecture that relies on a general-purpose host for PHY processing, certain delays are unavoidable. For example, there will always be a significant delay between when a packet is received and when the host can act on it. The reason is that the data needs to be streamed over an I/O bus (e.g. USB) and PHY layer processing must be completed. This means that there will always be a non-trivial latency associated with a receive-transmit packet sequence, e.g. RTS-CTS or DATA-ACK. This delay will shrink as technology improves (faster buses and CPUs).

VIII. RELATED WORK

We briefly discuss related work in three areas: SDRs, software frameworks for SDRs, and MAC protocols for SDRs.

Early SDRs include SPEAKeasy [12][13] and RDRN [14]. They were mostly designed to meet the requirements of the military, i.e. the rapid deployment of radio systems in mobile and dynamic environments. They had substantial dedicated computational power to support physical layer processing in software. Bose, et al., explored a different style of SDR: they use a relatively simple device and rely on an off-the-shelf shelf PC for processing [15]. This approach was commercialized by Vanu Inc, and has also been explored by others [16][17]. More recently, several research groups have developed SDR hardware for use in the NSF NeTS ProWin program. The hardware ranges from simple PC cards (USRP [20]) to more aggressive stand-alone hardware (KU Agile Radio [21]).

Developing software for SDRs is a challenging problem both because of the computational demands of PHY processing and the diversity of SDR hardware. As a result, people have developed environments that encourage modular software development, thus encouraging code reuse and code sharing. The US military developed the Joint Tactical Radio System (JTRS) Software Communications Architecture (SCA) as a standard [20]. Virginia Tech has developed an open source implementation of SCA (OSSIE [21]) and several companies market products that support SCA. We use the GNU Radio framework, which has support for the USRP.

A final area of related work is research on building MAC protocols for SDRs. Pant et al. [1] implemented a slotted ALOHA protocol with GNU Radio as the PHY. They replaced the MAC and PHY layers with a custom MAC based on slotted ALOHA and PHY using GNU Radio. The MAC and PHY communicate via UNIX domain sockets. Ethernet frames are passed between the network layer and MAC using the TAP/TUN interface. Holger von Malm [3] implemented a pure ALOHA and a send-and-wait ARQ protocol using the GNU Radio framework. The work also evaluates the performance of the system in terms of latency. All the above work reinforces the need for a platform to develop MAC protocols that support PHY processing. Finally, there is an active discussion on the GNU Radio mailing list, spearheaded by researchers from BBN, on how to extend GNU Radio to support various packet processing functions; several of these functions are similar to the ones we identified in Section VII. This approach is an alternative to the Click-based approach being explored in this paper. This approach has the potential of creating a highly streamlined framework, but there is a significant effort required to extend GNU Radio to support packet processing.

IX. Conclusion

In this paper we looked at the question of how to develop a framework that supports both MAC protocol and PHY layer development for software radios in an integrated fashion. To gain insight in the different design options, we used two existing frameworks to develop a MAC protocol for the USRP software radio. First, we used GNU Radio, which was originally designed to support PHY layer development. We found that while it possible to implement a basic MAC protocol in GNU Radio, some key protocol features were difficult or impossible to implement. Examples include timers and coordinated transmit-receive processing. We also used Click, a framework for protocol development. We discussed different ways of adding GNU Radio PHY blocks to Click and described an implementation in which a GNU Radio flow graph is encapsulated as a single Click element to implement PHY layer functionality. We found that this approach to integrating GNU Radio and Click provides a framework that makes it relatively easy to develop MAC/PHY protocol stacks for the USRP and we used a proof-of-concept TDMA MAC implementation to demonstrate this capability. Further research is needed to compare the performance of this approach with alternate designs.

Finally, we found that USRP and GNU Radio are very attractive platforms for experimenting with software radios. However, we identified a number of system-level challenges that require further study. Among these, offloading time-critical functions to the SDR device, the transfer time between the USRP and host computer, and the mixed kernel-user space architecture are the most critical.

Acknowledgment

We would like to thank Eric Blossom and Matt Ettus for helping us understand the GNU Radio framework and the USRP. We thank Eddie Kohler for helping us with Click. In addition, we are very grateful for the help we received from the many contributors to the GNU Radio and Click mailing lists.

Rahul Dhar is currently with Microsoft, Gesly George is currently with Amaranth, and Amit Malani is currently with Bloomberg.

References

[1] Volodymyr Kindratenko, Meenal Pant, David Pointer. Deploying the OLSR protocol on a network using SDR as the physical layer. NCASSR Technical Report, UCLA, May, 2005.

[2] Rahul Dhar. Towards an adaptive MAC layer – A CSMA/CA Scheme for the GNU Radio platform. Master's Thesis, INI, CMU, May 2005.

[3] H. von Malm. Implementing physical and data link control layer on the GNU software-defined radio platform. Bachelor's thesis, University of Paderborn, Computer Networks Group, December 2005.

[4] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router, ACM Transactions on Computer Systems **18**(3), August 2000, pages 263-297.

[5] Eddie Kohler. The Click modular router, Ph.D. thesis, MIT, November 2000.

[6] Click Timers. http://pdos.csail.mit.edu/click/doxygen/classTimer.html

[7] Eric Blossom. Re: [Discuss-gnuradio] USRP Synchronization. May 10, 2006

[8] David Lapsley. [Discuss-gnuradio] Proposed enhancements for data networking. 21 April, 2006

[9] Eric Blossom, Exploring GNU Radio, November 2004, available at http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html

[10] Gesly George. Exploring a platform for developing MAC protocols for Software Radios: Combining GNU Radio with Click, Master's Thesis, INI, CMU, May 2006.

[11] Amit Malani. Device Support for Software MAC Protocols, Master's Thesis, INI, CMU, May 2006.

[12] P. Cook and W. Bonser. Architectural overview of the speakeasy system. IEEE Journal on Selected Areas in Communications 17, 4 (Apr 1999), 650–661.

[13] R. Lackey and D. Upmal, D. Speakeasy: the military software radio. IEEE Communications Magazine 33, 5 (May 1995), 56–61.

[14] J. Evans, G. Minden, K. Shanmugan, G. Prescott, V. Frost, B. Ewy, R. Sanchez, C. Sparks, K. Malinimohan, J. Roberts, R. Plumb, and D. Petr. The rapidly deployable radio network. IEEE Journal on Selected Areas in Communications 17, 4 (Apr 1999), 689–703.

[15] V. Bose, M. Welborn, and J. Guttag. Virtual radios. IEEE Journal on Selected Areas in Communications 17, 4 (Apr 1999), 591–602.

[16] H. Shiba, Y. Shirato, H. Yoshioka, and I. Toyoda. Software De.ned Radio Prototype (I) – System Design and Performance Evaluation. NTT Technical Review 1, 4 (Jul 2003), 15–23.

[17] T. Shono and M. Matsui. Software De.ned Radio Prototype (II) – Implementation and Evaluation of IEEE 802.11 Wireless LAN. NTT Technical Review 1, 4 (Jul 2003), 24–30.

[18] Ettus Research LLC, The USRP, http://www.ettus.com/

[19] Gary Minden, KU Agile Radio Overview, available at https://www.csg.ethz.ch/education/lectures/sdm/KURadio_Overview_A50718.pdf

[20] JTRS, Software Communications Architecture Specification, version 2.2, November 2001, available from http://jtrs.spawar.navy.mil/sca/

[21] OSSIE, Open Source SCA Implementation: Embedded, available from http://ossie.mprg.org/

[22] Norman Hutchinson and Larry Peterson. The x-kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering, 17(1):64-76, Jan 1991.

[23] Magesh Kannan, Ed Komp, Gary Minden, and Joseph Evans. Design and Implementation of Composite Protocols. Technical Report ITTC-FY2003-TR-19740-05, Feb 2003.

[24] Michael Neufeld, Ashish Jain, Dirk Grunwald, Network protocol development with nsclick, Wireless Networks, 10(5):569-581, Sep 2004.

[25] Michael Neufeld, Ashish Jain, Dirk Grunwald**.** Nsclick:: bridging network simulation and deployment, Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems, Atlanta, 2002, pages 74 – 81.

[26] Universal TUN/TAP Driver. http://vtun.sourceforge.net/tun