TECHNICAL UNIVERSITY OF CRETE

Department of Electronic and Computer Engineering

Implementation and performance evaluation of the 802.11 CSMA/CA MAC protocol in SDR

Diploma Thesis

By

Panagiotis K.Matzakos

Submitted to the Department of Electronic & Computer Engineering in partial fulfillment of the requirements for the ECE Diploma Degree.

Advisor: Professor Liavas Athanasios

Co-advisor: Assistant Professor Karystinos Georgios

Co-advisor: Assistant Professor Mpletsas Aggelos

October 2010

To my parents and my brother for their deepest love and support.

Acknowledgements

I would like to express my gratitude to my advisor, Prof. Athanasios Liavas for the assignment of this thesis and his encouragement, guidance and support from the initial to the final level of its implementation. His wide knowledge and experience in the field of telecommunications meant a great deal to me. Deepest gratitude is also due to Assist. Prof. Aggelos Bletsas for his invaluable guidance and assistance.

Special thanks also to postgraduate student Manolis Matigakis without whose knowledge and assistance, this thesis would not have been successful.

Finally, i want to thank my beloved friends, who have always been there for me, standing by me as a family here in Chania.

Abstract

A software-defined radio system, or SDR, is a radio communication system where components that have been typically implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead implemented by means of software on a general purpose computer, providing much greater flexibility for the experimentation with different PHYs (physical layers) MACs (Medium Access Control) and other network layers as well as the interactions between them. A basic SDR system consists of a personal computer, Analog-to-Digital (AD) and Digital-to-Analog (DA) converters preceded by an RF front end.

In this thesis, we implemented and measured the throughput of the primary MAC technique of 802.11, called distributed coordination function (DCF), in an SDR system.

Contents

List of Figures

Chapter 1

Introduction

The 802.11 CSMA/CA *Distributed Coordination Function* (DCF) is a carrier sense multiple access with collision avoidance (CSMA/CA) scheme with binary slotted exponential backoff.

In our SDR, the RF front-end is a Universal Software Radio Peripheral (USRP) board. The PHY processing is performed in GNU Radio framework, which defines a flexible API for the USRP board, and the basic access scheme of the DCF is implemented in Click Modular Router framework. The carrier sensing mechanism of the protocol was implemented using Inter Process Communication between Click and GNU Radio processes. In particular, each packet identified by PHY processing (GNU Radio) is sent to MAC (Click Modular Router) using UNIX datagram sockets.

The large Rx-Tx, Tx-Rx turnaround times as well as the large packets' processing times for each station lead to a total delay in the scale of milliseconds. As a result, we had to limit to a suboptimal performance and approximate our own system's slot time, using a technique desribed in chapter 3.

Then, we evaluated our system's performance by measuring its normalized throughput. Our experiments were conducted with 2 and 4 stations communicating with a base station for various values of total offered load. Finally, the experimental results were compared to the theoretical ones as these are presented in [8].

Chapter 2

Software Tools and the USRP

In this chapter, we will describe the architecture and the operation principles of our RF front-end (i.e. USRP testbed) as well as the two frameworks we used in our SDR (i.e. GNU Radio framework for PHY and Click Modular Router for MAC implementation).

2.1 USRP

The Universal Software Radio Peripheral, or USRP, is designed to allow general purpose computers to function as high bandwidth software radios. In essence, it serves as a digital baseband and IF section of a radio communication system [1].

The basic design philosophy behind the USRP has been to do all of the waveform-specific processing, like modulation and demodulation, on the host CPU. All of the high-speed general purpose operations like digital up and down conversion, decimation and interpolation are done on the FPGA [1].

The true value of the USRP is in what it enables engineers and designers to create on a low budget and with a minimum of effort. A large community of developers and users have contributed to a substantial code base and provided many practical applications for the hardware and software. The powerful combination of flexible hardware, open-source software and a community of experienced users make it the ideal platform for software radio development [1].

2.1.1 USRP1

In our implementation we used the USRP1 devices, which are consisted of one mainboard and up to four daughterboards. With this design, it is easy for anyone to make his own system working in different frequencies simply by choosing the appropriate daughterboard. We used the RFX2400 daughterboars in the 2.4Ghz area.

RFX2400 daughterboard

As mentioned above, this daughterboard works in the 2.4Ghz band. This band consists of a continuous spectrum range of 100 Mhz (one of the areas of the ISM band). The daughterboard has one transmitter and one receiver on it.
The transmitter takes the baseband analog signal which comes from the mainboard and modulates it to the central frequency that we choose through the software (gnu radio). Pulse shaping is also implemented in the software. The output of the transmitter goes to a two-sided switch which is connected to the input of the transmitter from the one side and to the antenna plug from the other side. The side of the switch is controlled through the software, depending on whether we transmit or receive.
The receiver consists of an oscillator, whose frequency is controlled by the software, and a mixer which mixes the signal coming from the antenna with the sinusodial signal coming from the oscillator. The receiver is direct conversion so the baseband signal which is produced is sent to the mainboard for sampling [2]. In table 2.1 you can see the RFX2400 specifications.

	RFX2400 specs
Frequency	2.3-2.9 GHz
Tx Power	50+ mW(17dBm)
Noise figure	6-10dB

Table 2.1: RFX2400 specs

Mainboard

The USRP has 4 high-speed analog to digital converters (ADCs), each at 12 bits per sample, 64MSamples/sec. There are also 4 high-speed digital to analog converters (DACs), each at 14 bits per sample, 128MSamples/sec. These 4 input and 4 output channels are connected to an Altera Cyclone EP1C12 FPGA. The FPGA, in turn, is connected to a USB2 interface chip, the Cypress FX2, and to the computer. The USRP is connected to the computer via a high speed USB2 interface.
So, in principle, we have 4 input and 4 output channels if we use real sampling. However, we can have more flexibility (and bandwidth) if we use complex (IQ) sampling. Then we have to pair them up, so we get 2 complex inputs and 2 complex outputs [1].

ADC

There are 4 high-speed 12-bit AD converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz.
The full range of the ADCs is 2V peak to peak, and the input is 50 ohms differential. This is 10mW, or 10dBm. There is a programmable gain amplifier (PGA) before the ADCs which amplifies the input signal so that it utilizes the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20dB. With gain set to zero, full scale inputs are 2 Volts peak-to-peak differential. When set to 20 dB, only .2 V p-p differential input signal is needed to reach full scale. This PGA is software programmable [1].

DAC

At the transmit path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, we will probably want to stay below it to make filtering easier. A useful output frequency range is from DC to about 44MHz. The DACs can supply 1V peak to a 50 ohm differential load, or 10mW (10dBm). There is also PGA used after the DAC, providing up to 20dB gain. This PGA is software programmable. The DAC signals

($IOUTP_A/IOUTN_A$ and $IOUTP_B/IOUTN_B$) are current-output, each varying between 0 and 20 mA. They can be converted into differential voltages with a resistor [1].

FPGA

According to the above, the information rate that a USRP sends and receives at the same time is:

$$(64MSPS * 12bit/Sample + 128MSPS * 14bit/Sample) * 2 = 640Mbyte/sec. \quad (2.1)$$

But the data rate tha the USB port can support is up to 32Mbyte/sec. Moreover the samples should be transformed from 12 and 14 bits to the closest multiple of 8 bits. These two processes are undertaken by the FPGA [2].
On the FPGA, there are two digital filters (**decimation** and **interpolation**). The decimation filter's input is the flow of the samples from the ADC which comes with a rate of 64MS/s. The decimation filter subsamples its input flow by a decimation rate factor, which is chosen through the software. The minimum decimation rate is 4 and the maximum 256. The decimation rate sets the limit of the bandwith around the central frequency at which the receiver listens to [2].
Accordingly, the interpolation filter oversamples the data flow by an **interpolation rate** factor, which is also chosen through the software, so that the final sampling rate is 128MS/s (sampling rate of the DAC). Finally, as the data rate of the USB is not stable, we need a buffer before the interpolation filter in which the samples coming from the PC are stored and they are read by a rate of 128MSPS/interpolation so that in the end we have a 128MS/s [2].

2.2 GNU Radio

GNU Radio is a free software development toolkit that provides the signal processing runtime and processing blocks to implement software radios using readily-available, low-cost external RF hardware and commodity processors. It is widely

used in hobbyist, academic and commercial environments to support wireless communications research as well as to implement real-world radio systems [3].

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. Thus, the developer is able to implement real-time, high-throughput radio systems in a simple-to-use, rapid-application-development environment [3].

While not primarily a simulation tool, GNU Radio does support development of signal processing algorithms using pre-recorded or generated data, avoiding the need for actual RF hardware [3].

To compose an application in GNU Radio, users first build signal processing blocks in C++ and then connect these blocks together using the Python language to form a flow graph. In addition to the relative ease of programming in C++, this approach makes it easy to transition blocks that are part of a PHY simulator into GNU Radio [4].

2.3 The Click Modular router

Click is a new software architecture for building flexible and configurable routers.[1] A Click router is assembled from packet processing modules called elements. Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Several features make individual elements more powerful and complex configurations easier to write, including pull connections, which model packet flow driven by transmitting hardware devices, and flow-based router context, which helps an element locate other interesting elements.

First, packet handoff along a connection may be initiated by either the source end (push processing) or the destination end (pull processing). This cleanly models most router packet flow patterns, and pull processing makes it possible to write

[1] Copyright for this section belongs to [5].

composable packet schedulers. Second, the flow-based router context mechanism lets an element automatically locate other elements on which it depends; it is based on the observation that relevant elements are often connected by the flow of packets. These features make individual elements more powerful and configurations easier to write.

2.3.1 Architecture

A Click router configuration is a directed graph whose nodes are called elements. A single element represents a unit of router processing. An edge, or connection, between two elements represents a possible path for packet transfer. This graph resembles a flowchart, except that connections represent packet flow, not control flow, and elements are actual objects that may maintain private state. Inside a running router, each element is a C++ object and connections are pointers to elements. The overhead of passing a packet along a connection is a single virtual function call. The most important properties of an element are:

- *Element class.* Like objects in an object-oriented program, each element has a class that determines its behavior.
- *Input and output ports.* Ports are the endpoints of connections between elements. An element can have any number of input or output ports, which can have different semantic meanings (a normal and an error output, for example).
- *Configuration string.* Some element classes support additional arguments, used to initialize per-element state and fine-tune element behavior. The configuration string contains these arguments.

Figure 2.1 shows how we diagram these properties for a single element, Tee(2).Tee is the element class; a Tee copies each packet it receives from its single input port, sending one copy to each output port. (The packet data is not copied: Click packets are copy-on-write.) Configuration strings are enclosed in parentheses: the 2 in Tee(2) is a configuration string that Tee interprets as a request for two outputs. Every action performed by a Click routers software is encapsulated in an element, from device

reading and writing to queueing, routing table lookups, and counting packets. The user determines what a Click router does by choosing the elements to be used and the connections among them. Figure 2.2 shows a sample router that counts incoming packets, then throws them all away. Click provides two kinds of connections between elements, push and pull. In a push connection, the upstream element hands a packet to the downstream element; in a pull connection, the downstream element asks the upstream element to return a packet. Each kind of handoff is implemented as a virtual function call. Packet arrival usually initiates push processing, which stops when an element discards the packet or stores it for later. Output interfaces initiate pull processing when they are ready to send a packet; processing flows backwards through the graph until an element yields up a packet. Pull elements can simply and explicitly represent decisions that should occur at packet transmission time, such as packet scheduling.

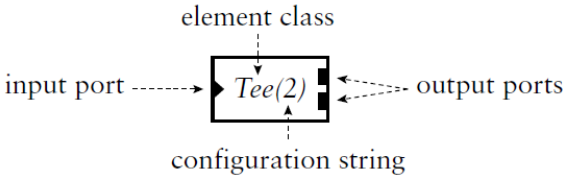

Figure 2.1: A sample element. Triangular ports are inputs and rectangular ports are outputs.(Figure from [5])

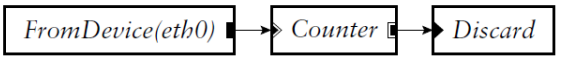

Figure 2.2: A router configuration that throws away all packets.(Figure from [5])

2.3.2 Control flow and queues

When an element receives a packet from a push connection, it must store it, discard it, or forward it to another element for more processing. Most elements forward packets by calling the next element's push function. Since packet handoff is just a virtual function call, a Click CPU scheduler can not stop packet processing at arbitrary points, elements must cooperatively choose to stop processing. Packet

storage must be implemented by the element itself. Click elements do not have implicit queues on their input and output ports, or the associated performance and complexity costs. Instead, Click queues are explicit objects, implemented by a separate element (Queue). This enables valuable configurations that are difficult to arrange otherwise - for example, a single queue feeding multiple interfaces, or a queue feeding a traffic shaper on the way to an interface. Queue is the most common element that stops packet processing, giving the system a chance to schedule different work: it enqueues packets it receives rather than passing them on. Thus, the placement of Queues in a configuration determines that configuration's execution profile. If a user wants to carefully manage packet scheduling as soon as packets enter the system, he must put Queues early in the graph.

2.3.3 Push and pull processing

Click supports two kinds of connections, push and pull. On a push connection, packets start at the source element and are passed downstream to the destination element. This corresponds to the way packets move through most software routers. On a pull connection, in contrast, the destination element initiates packet transfer: it asks the source element to return a packet, or a null pointer if no packet is available. This is the dual of a push connection. Each of these forms of packet transfer is implemented by one virtual function call.

The processing type of a connection—whether it is push or pull—is determined by the ports at its endpoints. Each port in a running router is either push or pull; connections between two push ports are push, and connections between two pull ports are pull. Connections between a push port and a pull port are illegal. Elements set their ports' types as the router is initialized. They may also create agnostic ports, which behave as push when connected to push ports and pull when connected to pull ports. When a router is initialized, the system propagates constraints until every agnostic port has been assigned to either push or pull. In our configuration diagrams, black ports are push and white ports are pull; agnostic ports are shown as push or pull ports with a double outline. Figure 2.3 shows how push and pull work in a simple router.

Push processing is appropriate when unsolicited packets arrive at a Click router—for example, when packets arrive from a device. The router must handle such packets as they arrive, if only to queue them for later consideration. Pull processing is appropriate when the Click router needs to control the timing of packet processing. For example, a router may transmit a packet only when the transmitting device is ready. In Click, transmitting devices are elements with one pull input; they therefore initiate packet transfer, and can ask for packets only when they are ready.

Pull processing also models the scheduling decision inherent in choosing the next packet to send. A Click packet scheduler is simply an element with one pull output and multiple pull inputs. Such an element responds to a pull request by choosing one of its inputs, making a pull request to that input, and returning the packet it receives. (If it receives a null pointer, it will generally try another input.) These elements make only local decisions: different scheduling behaviors correspond to different algorithms for choosing an input. Thus, they are easily composable.

The following properties hold for all correctly configured routers: Push outputs must be connected to push inputs, and pull outputs must be connected to pull inputs. Each agnostic port must be used as push or pull exclusively. Furthermore, if packets arriving on an agnostic input might be emitted immediately on one of that element's agnostic outputs, then both input and output must be used in the same way (either push or pull). Finally, push outputs and pull inputs must be connected exactly once. This ensures that each packet transfer request—either pushing to an output port or pulling from an input port—is along a unique connection. These properties are automatically checked by the system during router initialization. Figure 2.4 demonstrates some property violations.

These properties are designed to catch intuitively invalid configurations. For example, the connection in Figure 2.4 from FromDevice to ToDevice is illegal because FromDevice's output is push while ToDevice's input is pull. But this connection is intuitively illegal, since it would mean that ToDevice might receive packets when it was not ready to send them. The Queue element, which converts from push to pull, also provides the temporary packet storage this configuration requires.

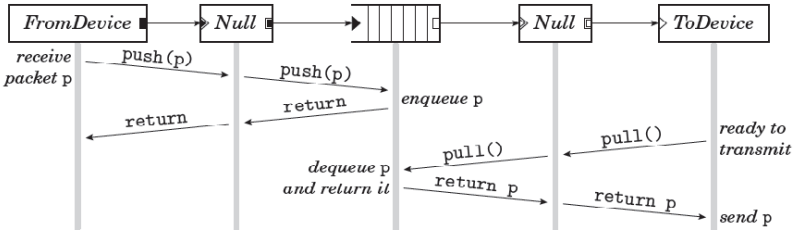

Figure 2.3: Push and pull control flow.This diagram shows functions called as a packet moves through a simple router; time moves downwards. The central element is a Queue. During the push, control flow moves forward through the element graph starting at the receiving device; during the pull, control flow moves backwards through the graph, starting at the transmitting device. The packet p always moves forward.(Figure from [5])

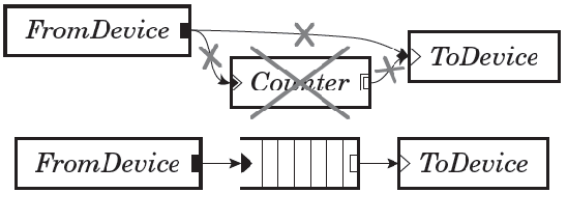

Figure 2.4: Some push and pull violations. The top configuration has four errors: (1) FromDevice's push output connects to ToDevice's pull input; (2) more than one connection to FromDevice's push output; (3) more than one connection to ToDevice's pull input; and (4) an agnostic element, Counter, in a mixed push/pull context. The bottom configuration, which includes a Queue, is legal. In a properly configured router, the port colors on either end of each connection will match.(Figure from [5])

2.3.4 Language

Click configurations are written in a simple language with two important constructs: declarations create elements, and connections say how they should be connected. Its syntax is easy enough to learn from an example; Figure 2.5 uses it to define a trivial router.

```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// ... and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Figure 2.5: Two Click-language definitions for the trivial router of Figure 2.2. (Figure from [5])

Chapter 3

Implementation

In this chapter, we will first discuss how the PHY and the MAC layer were connected through IPC (*Inter Process Communication*), then we will describe the experimental setup and finally the DCF of the 802.11 CSMA/CA MAC protocol and how this was implemented through the Click Modular router.

3.1 IPC between Click and GNU Radio processes

This approach discussed in [6] suggests that Click and GNU Radio execute as individual processes with Click implementing our MAC Protocol and GNU Radio performing the PHY processing. So each station (terminal) in our implementation has three running processes, one for Click and two for GNU Radio (sender and receiver). Thus, the two GNU Radio processes communicate with Click, by sending and receiving messages through UNIX domain datagram sockets.

Sockets are based on the client-server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. The client needs to be aware of the existence and the address of the server, but the server does not need to know the address (or even the existence) of the client before the connection has been established. A socket is one end of an interprocess communication channel. Each of the two processes establish their own socket [7].

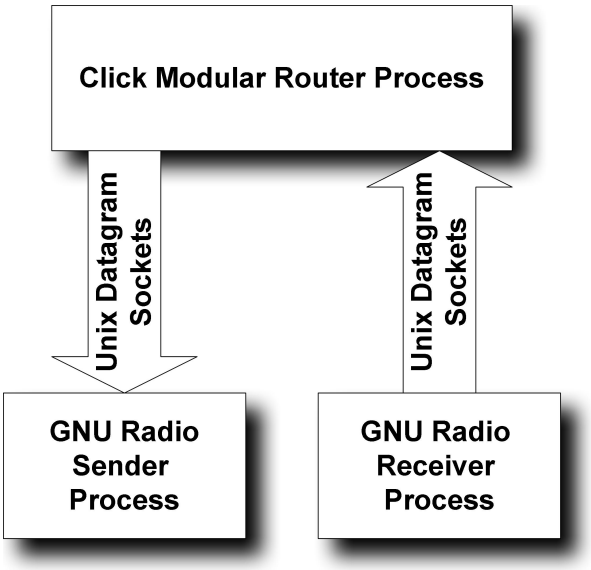

Figure 3.1: IPC using UNIX datagram sockets

In our case, the packets generated at Click flow to the GNU Radio sender process through a socket and conversely, the packets received from the PHY flow from the GNU Radio receiver process to Click for MAC processing, as shown in figure 3.1. In the first case, the GNU Radio process is the server which creates a socket through the **socket()** system call and binds it to a specific address (entry in the filesystem) through the **bind()** system call. Then, it can receive messages (packets) from the Click process using the **recvfrom()** system call. The Click process is the client which also creates its own socket through **socket()** system call and can send messages (packets) to the GNU Radio process's socket through **sendto()** system call. The opposite process takes place in the second case (i.e, the GNU Radio process is the client and the Click process is the server).

3.2 Experimental Setup

In this section, we will follow the route of each generated packet from the MAC Layer of the sender (station) to the MAC Layer of the receiver (base station) and show which actions are performed in each stage.

The Station's Click Process

The router configuration of the MAC Layer can be viewed in Figure 3.2. **Rated-Source** is a standard Click Element which generates packets at specified rate. The generated packets are stored in the **Queue** that follows. Next to the Queue there is the main element of our configuration which implements the MAC algorithm and sends the packets to the PHY layer according to it. When a packet is successfully acknowledged from the base station it passes to **Discard** standard element which drops it and initiates the next packet transfer.

The GNU Radio Sender Process

The packets sent to GNU Radio are received through the socket inside a packet source block. This block converts the bytes of each message to 4-QAM symbols. It also adds the training symbols, the MAC address of the particular station, the MAC address of the base station and the frame number of the current packet. The processed packet is then oversampled by the next block, passed through a SRRC (Square Root Raised Cosine) transmit filter and finally flows to usrp sink block which sends the packet to the USRP for transmission at a specified frequency.

The GNU Radio Receiver Process

In the GNU Radio Receiver Process the data arriving at the USRP's RF front-end (tuned through GNU Radio at the specified frequency) are converted to baseband signal and they are sent to USRP source block. Then they pass through a SRRC receiver filter and after that through the packetizer block. The above detects the packets inside the data stream. Frequency offset estimation and cancelation, channel estimation and equalization take place afterwards resulting in retrieving each packet. The next block converts the 4-QAM symbols to bytes and sends each packet to MAC Layer (Click process).

The Base Station's Click Process

After having received the packet from its socket this process reads the transmitter's MAC address as well as the frame number of the current packet and inserts them to the Acknowledgment packet which is generated. The above follows the opposite route to reach the initial station.

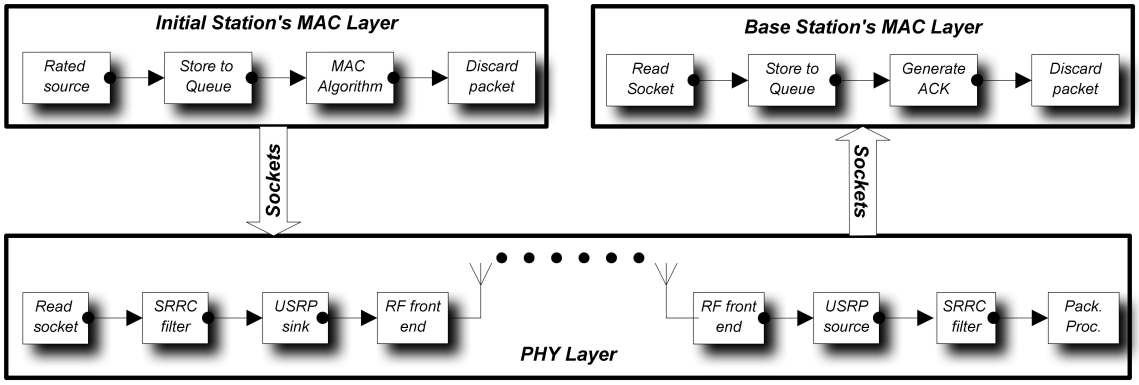

Figure 3.2: Experimental Setup

3.3 802.11 Distributed Coordination Function

According to the DCF, a station with a new packet to transmit monitors the channel activity. If the channel is idle for a period of time equal to a distributed interframe space (DIFS), the station transmits. Otherwise, if the channel is sensed busy (either immediately or during the DIFS), the station persists to monitor the channel until it is measured idle for a DIFS. At this point, the station generates a random backoff interval before transmitting (this is the Collision Avoidance feature of the protocol), to minimize the probability of collision with packets being transmitted by other stations. In addition, to avoid channel capture, a station must wait a random backoff time between two consecutive new packet transmissions, even if the medium is sensed idle in the DIFS time [8].

For efficiency reasons, DCF employs a discrete-time backoff scale. The time immediately following an idle DIFS is slotted, and a station is allowed to transmit only at the beginning of each slot time. The slot time size is set equal to the time needed at any station to detect the transmission of a packet from any other

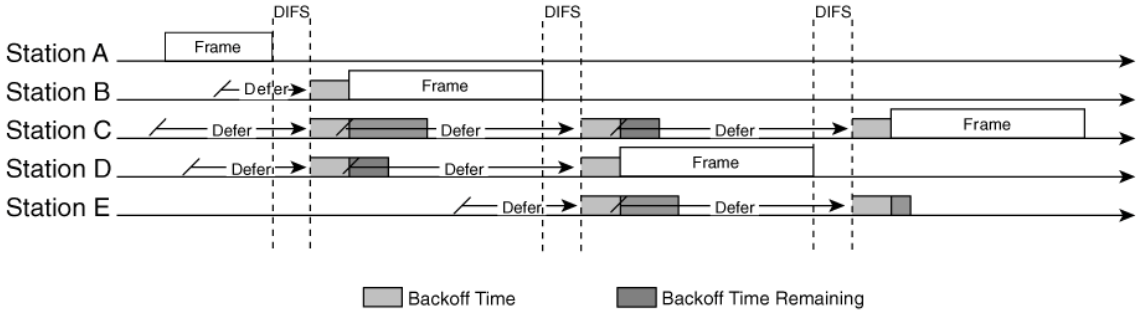

Figure 3.3: IEEE 802.11 DCF Transmission Example (Figure from [11])

station. It accounts for the propagation delay, for the time needed to switch from the receiving to the transmitting state (RX-TX-Turnaround Time), and for the time to signal to the MAC layer the state of the channel (busy detect time) [8].

DCF adopts an exponential backoff scheme. At each packet transmission, the backoff time is uniformly chosen in the range $(0, W-1)$. The value w is called contention window, and depends on the number of transmissions failed for the packet. At the first transmission attempt, w is set equal to a value CWmin called minimum contention window. After each unsuccessful transmission, w is doubled, up to a maximum value $CWmax = 2^m * CWmin$ [8].

The backoff time counter is decreased as long as the channel is sensed idle, frozen when a transmission is detected on the channel, and reactivated when the channel is sensed idle again for more than a DIFS. The station transmits when the backoff time reaches zero [8].

Since the CSMA/CA does not rely on the capability of the stations to detect a collision by hearing their own transmission, an ACK is transmitted by the destination station to signal the successful packet reception. The ACK is immediately transmitted at the end of the packet, after a period of time called short interframe space (SIFS). As the SIFS (plus the propagation delay) is shorter than a DIFS, no other station is able to detect the channel idle for a DIFS until the end of the ACK. If the transmitting station does not receive the ACK within a specified ACK-Timeout, or it detects the transmission of a different packet on the channel, it reschedules the packet transmission according to the given backoff rules [8].

Figure 3.3 shows a simplified example of how the DCF process works. (In this

simplified DCF example, no acknowledgments are shown and no fragmentation occurs.) The DCF steps illustrated in Figure 3.3 work as follows:

1. Station A successfully sends a frame, and three other stations also want to send frames but must defer to Station A's traffic.

2. When Station A completes transmission, all the stations must still defer to the DIFS. When the DIFS is complete, stations that want to send a frame can begin decreasing their backoff counters (decreasing by one for every slot time that passes). If their backoff counters reach 0 and the channel is available, they may send their frame.

3. Station B's backoff counter reaches 0 before Stations C and D, so Station B begins transmitting its frame.

4. When Stations C and D detect that Station B is transmitting, they must stop decreasing their backoff counters and again defer until the frame is transmitted and a DIFS has passed.

5. During the time that Station B is transmitting a frame, Station E gets a frame to transmit, but because Station B is sending a frame, Station E must defer in the same manner as Stations C and D.

6. When Station B completes transmission and the DIFS has passed, stations with frames to send begin decreasing their backoff counters again. In this case, Station D's backoff counter reaches 0 first, and the station begins transmission of its frame.

7. The process continues as traffic arrives on different stations.

3.3.1 Pseudocode

At this point we present our pseudocode of 802.11 DCF (Algorithm 1) to show how we implemented the various steps of the MAC algorithm under Click Modular Router platform.

Line 1 indicates that after each packet's successful transmission, a pull request is made on the queue to get the next packet, if available, and that the contention window length is set to its minimum value. If the queue is empty, we are in the case of non-consecutive transmissions (i.e there is an empty queue incident between two new packet transfers). This means that the current station will not necessarily enter the backoff stage of the algorithm the next time it has a packet available for transmission (**backoff_stage = false**). On the contrary, as mentioned in the above description of the DCF, if no empty queue incident occurs between two new packet transfers the station has to enter the backoff stage to avoid channel capture (**backoff_stage = true**).

Having a packet available for transmission, each station has to listen to its socket for a DIFS interval. If the socket receives a message during that time (i.e. the channel got busy during DIFS period), then the station has to secure that the socket remains empty for a DIFS period and then enter the backoff stage. If not, plus the current transmission is not consecutive with the previous one, the station is free to transmit (**backoff_stage = false**).

Lines 11-22 implement the exponential backoff stage of the algorithm. The station picks a random backoff time in line 12 and then it listens to each socket for that time through **select()** system call. select() returns 0 when no message was found in the socket during the specified time and 1 immediately after a message was found in the socket. The two timestamps, before and after the select() system call are used to calculate the elapsed time during which the socket was empty (i.e. the channel was sensed idle). Thus, every time the backoff timer is frozen due to a detected transmission we can calculate the remaining time, as shown in line 20. Lines 17-19 indicate that once the backoff timer is frozen, the station has to wait for its socket being empty for a DIFS interval before it is allowed to continue the countdown.

After the backoff stage, the station is free to transmit its packet. Then, it listens to its socket for an ACK_Timeout period, waiting to receive an Acknowledgment from the base station. At this point, the transmission is considered to be unsuccessfull if one of the following incidents occurs:

1. we did not receive anything from the socket during ACK_Timeout,

2. we received an erroneous packet,

3. the receiver's MAC address of the ACK packet is not that of the current station.

In the case of an unsuccessful transmision, the contention window value is increased, unless it has already reached its maximum value. Then, the station re-enters the backoff stage. If nothing from the above occurs and the transmission is successful, the current packet is discarded and a new pull connection from the queue takes place.

Algorithm 1 802.11 DCF

```
1: p = pull next packet from queue, cw_length = CW_MIN
2: if p == null then
3:     backoff_stage = false {case of non consecutive transmissions}
4:     return p
5: else
6:     wait for difs idle channel
7:     if channel got busy during DIFS then
8:         backoff_stage = true
9:     end if
10:    repeat
11:        if backoff_stage == true then
12:            backoff = random interval from range (0, cw_length − 1) × slot
13:            while backoff > 0 do
14:                t1 = timestamp1
15:                rc = select(sock1,backoff)
16:                t2 = timestamp2
17:                if rc > 0 then
18:                    wait for DIFS idle channel
19:                end if
20:                backoff = backoff - (t2 - t1)
21:            end while
22:        end if
23:        send packet to PHY socket
24:        rc = select(sock2,ACK_Timeout)
25:        receive ACK from MAC socket
26:        if rc == 0 or erroneous packet or wrong MAC then
27:            if cw_length < CW_MAX then
28:                cw_length = 2 × cw_length
29:            end if
30:        else
31:            successfull transmission
32:        end if
33:        backoff_stage = true
34:    until successfull transmission
35: end if
36: return p
```

3.4 Other implementation issues

In this section we will discuss some implementation issues that we consider important and the way we dealt with them.

3.4.1 Software Constraints

As the time in our MAC protocol is slotted, it is of great importance to be very precise with the slot's duration and the other time spaces which depend on it (i.e DIFS, SIFS, ACK-Timeout).

Unfortunately, the GNU Radio-USRP interface results in great delays, which do not permit us to achieve as large data rates as those described in the 802.11 standard. This suboptimal performance is mainly due to the large Rx-Tx and Tx-Rx turnaround times for each tranceiver as well as the processing times for each packet. These two processes lead to a total delay in the scale of milliseconds.

As we mentioned in the description of the DCF, the slot time is defined as:

$$Slot = Rx\_Tx\_Turnaround\_Time + propagation\_delay + busy\_detect\_time \quad (3.1)$$

In order to calculate this duration directly, we should have had the same clock between a station and the base station and calculate the elapsed time from the generation of a packet in a station until the MAC layer notification of the base station. As this is impossible we used another way. Specifically, we calculated the slot time through the DIFS. The DIFS is defined as follows:

$$DIFS = 2 \times slot\_time + SIFS. \quad (3.2)$$

In our case, and since the Acknowledgments and the packets are of the same length (i.e. same propagation delay), the above definition means that the DIFS is equal to the time needed from the instance that we generate a packet until we receive its Acknowledgment (this time can be easily calculated in the same terminal). It is obvious from equation (3.2) that since we know the DIFS we can calculate the slot time as follows:

$$slot\_time = \frac{DIFS - SIFS}{2}. \quad (3.3)$$

3.4.2 Operation Frequency

As mentioned in Section 3.1, two gnu radio processes (transmitter and receiver) run in each station (including the base station). Since our whole system works in one central frequency, each station listens to its own packets as well. In order to deal with this, we read each packet's sender MAC Address and if it is the same with the current station's we discard the packet and don't pass it to the MAC layer.

In Table 3.1 we summarise our system's parameters and in Figure 3.4 we depict our MAC frame.

	System Parameters
Slot time	3.0 msec
DIFS	7.0 msec
ACK_Timeout	20 msec
SIFS	1.0 msec
Packet_length	400 bits
Tx_Power	31+ mw (15dbm)
Constellation scheme	4-QAM

Table 3.1: System Parameters

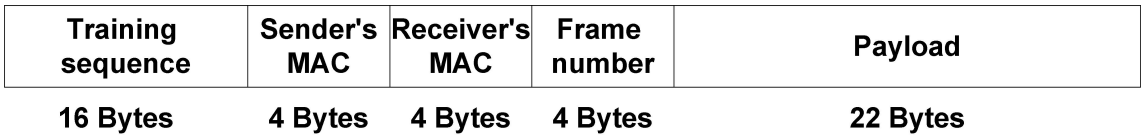

Figure 3.4: MAC frame

Chapter 4

Throughput Analysis

In this chapter we will analyze the performance of our system. Due to the runtime delays we mentioned in the previous chapter, our system cannot use the channel capacity efficiently and achieve large data rates. As a result, to analyze the system's throughput we first had to measure its capacity. Then, we measured the normalized throughput. We will discuss these issues in detail and present our results, after we see the 802.11 DCF's theoretical and simulation performance as this is presented by Giuseppe Bianchi in [8].

4.1 System Capacity

To measure the system's capacity, we counted the average number of acknowledged packets per second in a point-to-point link and stable conditions (i.e. packet error rate below 5 per cent). At this point, we took the acknowledgments into account because they bring on the same runtime delays with the packets (same Rx-Tx, Tx-Rx Turnaround times, same processing and propagation delays). Thus, we can not ignore them in the measurement of the system's capacity. In Figure 4.1, you can see the offered load versus tha packet error rate and the average number of acknowledged packets per second in a point-to-point link.

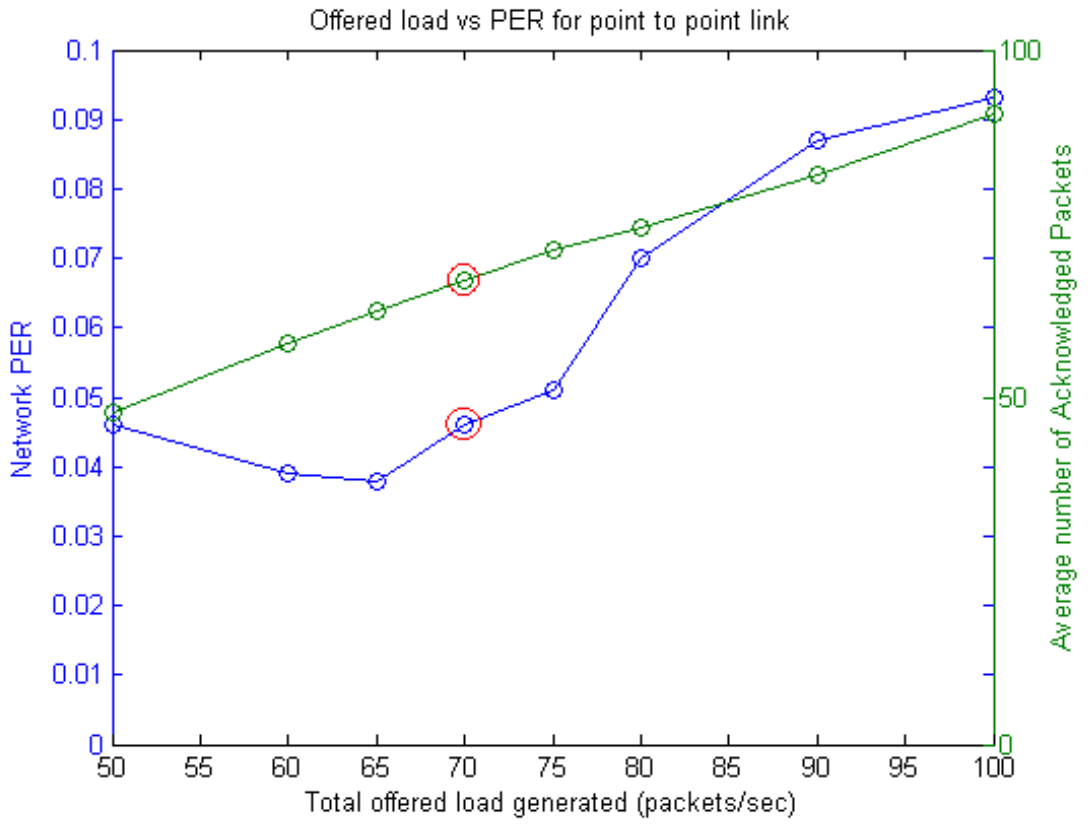

Figure 4.1: point-to-point link performance

As you can see, the upper bound of the offered load ($PER < 0.05$) is 70 packets/second for which we have an average of 66.76 acknowledged packets/sec. This is considered as our system's capacity.

4.2 Bianchi's simulation and theoretical results

4.2.1 Maximun and Saturation Throughput

Saturation throughput is defined as the limit reached by the system throughput as the offered load increases, and represents the maximum load that the system can carry in stable conditions [8].

As a random access scheme, the DCF of 802.11 exhibits an unstable behavior. Particularly, as the offered load increases, the throughput grows up to a maximum value known as "maximum throughput". Further increases of the offered load,

though, lead to a significant decrease of the system throughput. As a result, it is impossible to operate the random access scheme at its maximum throughput for a long period of time [8].

In Figure 4.2 (from [8]), we can see simulation results for 20 stations, which visualize this unstable behaviour. As described in [8], the offered load linearly increases with simulation time. The straight line represents the ideal offered load, normalized with respect of the channel capacity. The simulated offered load has been generated according to a Poisson arrival process of fixed size packets, where the arrival rate has been varied throughout the simulation to match the ideal offered load. The throughput was measured over 20 sec time intervals and normalized with respect to channel rate. We observe that the measured throughput follows closely the offered load for the first 260 s of simulation, while it asymptotically drops to the value 0.68 in the second part of the simulation run. This asymptotic throughput value is referred to as saturation throughput, and represents the system throughput in overload conditions. Note that, during the simulation run, the instantaneous throughput temporarily increases over the saturation value (up to 0.74 in the example considered), but ultimately it decreases and stabilizes to the saturation value. Queue build-up is observed in such a condition.

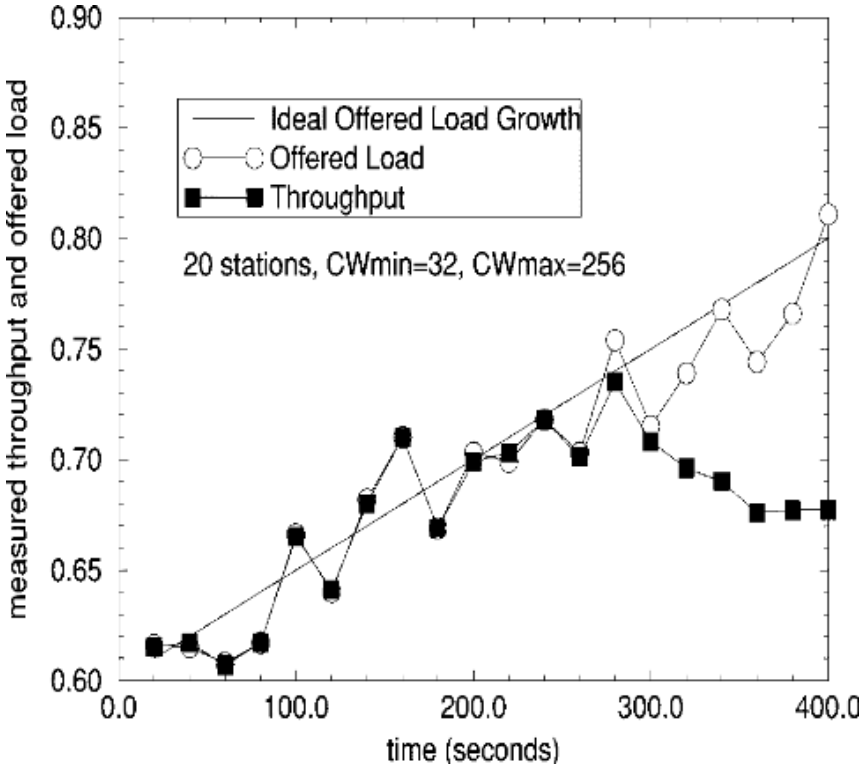

Figure 4.2: Measured Throughput with slowly increasing offered load (Figure from [8])

4.2.2 Bianchi's Model for performance evaluation

In [8], Bianchi describes a model for the analytical evaluation of the saturation throughput of 802.11 DCF[1]. According to that model, Bianchi extracts the stationary probability τ that a station transmits a packet in a randomly chosen time slot. Then, by studying the events that can occur in a randomly chosen time slot, he expresses the throughput as a function of the computed value τ.

Packet Transmission Probability

In saturation conditions, each station has immediately a packet available for transmission, after the completion of each successfull transmission. As a result, each packet needs to wait a random backoff time before transmitting.

Let $b(t)$ be the stochastic process representing the backoff time counter for a given

[1]copyrights for this subsection belong to [8].

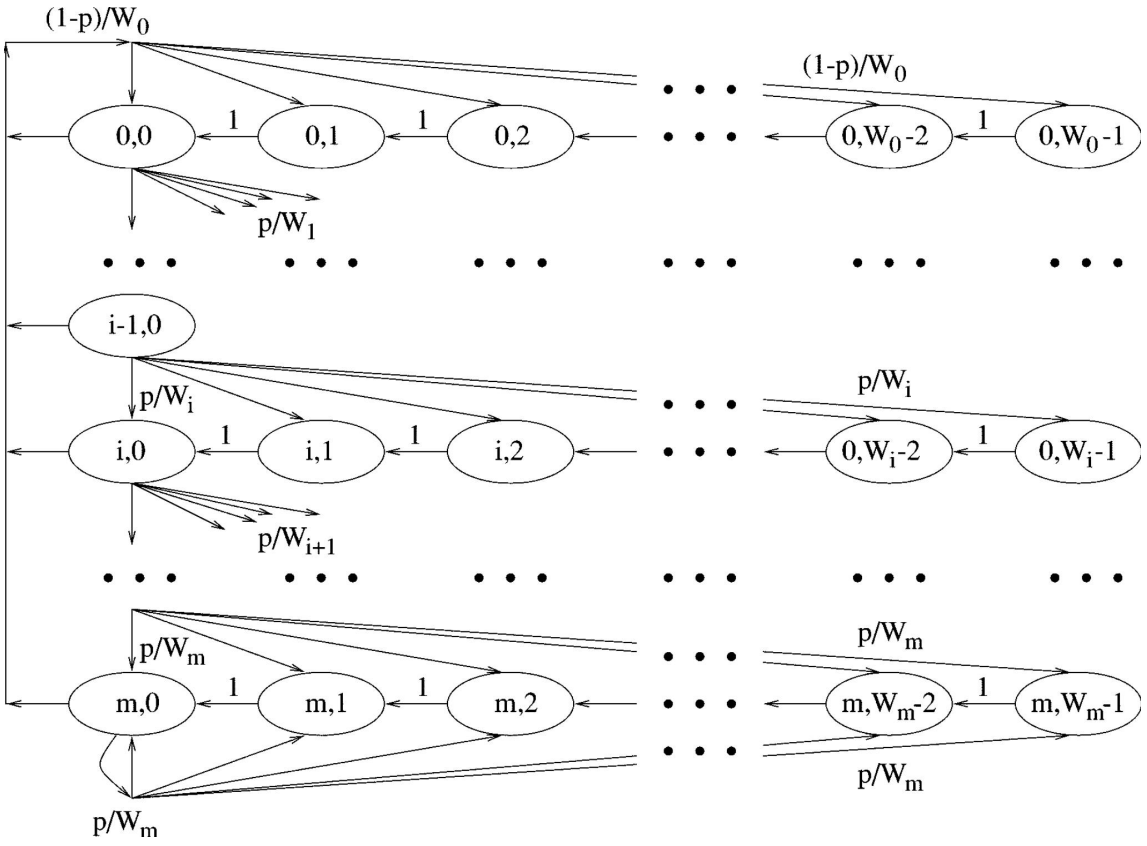

Figure 4.3: Markov chain for the backoff window size (Figure from [8])

station, where t and $t+1$ correspond to the beginning of two consecutive slot times. The backoff time counter of each station is decreased at the beginning of each time slot. We define the minimum value of the contention window, $W = CW_{min}$, and the maximum backoff stage, m, such that $CW_{max} = 2^m \times W$. We adopt the notation $W_i = 2^i \times W$, where $i \in (0, m)$. Let $s(t)$ be the stochastic process representing the backoff stage $(0, ..., m)$ of the station at time t.

The key approximation of this model is that at each transmission attempt, and regardless of the number of retransmissions suffered, each packet collides with constant and independent probability p.

Once independence is assumed, and p is supposed to be a constant value, it is possible to model the bidimensional process $\{s(t), b(t)\}$ with the discrete-time Markov chain depicted in Figure 4.3. In this Markov chain, the only non-null one-step transition probabilities are:

$$\begin{cases} P\{i,k\,|i,k+1\}=1 & k\in(0,W_i-2) \quad i\in(0,m) \\ P\{0,k\,|i,0\}=(1-p)/W_0 & k\in(0,W_0-1) \quad i\in(0,m) \\ P\{i,k\,|i-1,0\}=p/W_i & k\in(0,W_i-1) \quad i\in(1,m) \\ P\{m,k\,|m,0\}=p/W_m & k\in(0,W_m-1). \end{cases} \tag{4.1}$$

The first equation in (4.1) accounts for the fact that, at the beginning of each time slot, the backoff time is decreased. The second equation accounts for the fact that a new packet following a successful packet transmission starts with backoff stage 0, and thus the backoff is initially uniformly chosen from the range $(0, W_0-1)$. The third equation accounts for the fact that after an unsuccessful transmission at backoff stage $i-1$, the backoff stage is increased, and the new initial backoff value is uniformly chosen in the range $(0, W_i)$. Finally, the fourth case models the fact that once the backoff stage reaches the value m , it is not increased in subsequent packet transmissions.

Let $b_{i,k} = \lim_{t\to\infty} P\{s(t)=i, b(t)=k\}$, $i\in(0,m)$, $k\in(0,W_i-1)$ be the stationary distribution of the chain. It is easy to obtain a closed-form solution for this Markov chain. First note that

$$\begin{aligned} &b_{i-1,0}\cdot p = b_{i,0} \rightarrow b_{i,0} = p^i b_{0,0}, \quad 0<i<m \\ &b_{m-1,0}\cdot p = (1-p)b_{m,0} \rightarrow b_{m,0} = \tfrac{p^m}{1-p} b_{0,0}. \end{aligned} \tag{4.2}$$

Owing to the chain regularities, for each $k\in(1,W_i-1)$, it is

$$b_{i,k} = \frac{W_i-k}{W_i}\cdot \begin{cases} (1-p)\sum_{j=0}^{m} b_{j,0} & i=0 \\ p\cdot b_{i-1,0} & 0<i<m \\ p\cdot(b_{m-1,0}+b_{m,0}) & i=m. \end{cases} \tag{4.3}$$

Due to (4.2) and making use of the fact that $\sum_{i=0}^{m} b_{i,0} = b_{0,0}/(1-p)$, (4.3) can be written as

$$b_{i,k} = \tfrac{W_i-k}{W_i} b_{i,0} \overset{(4.2)}{\rightarrow} b_{i,k} = \tfrac{W_i-k}{W_i} p^i b_{0,0} \qquad i\in(0,m), \qquad k\in(0,W_i-1) \tag{4.4}$$

Thus, we have expressed all the values $b_{i,k}$ as functions of the value $b_{0,0}$ and of the conditional collision probability p. Finally, $b_{0,0}$ is determined by imposing the

normalization condition, that simplifies as follows:

$$\begin{aligned} 1 &= \sum_{i=0}^{m} \sum_{k=0}^{W_i-1} b_{i,k} = \sum_{i=0}^{m} b_{i,0} \sum_{k=0}^{W_i-1} \frac{W_i - k}{W_i} = \sum_{i=0}^{m} b_{i,0} \frac{W_i + 1}{2} \\ &= \frac{b_{0,0}}{2} \left[W \left(\sum_{i=0}^{m-1} (2p)^i + \frac{(2p)^m}{1-p} \right) + \frac{1}{1-p} \right] \end{aligned} \tag{4.5}$$

from which we obtain that,

$$b_{0,0} = \frac{2(1-2p)(1-p)}{(1-2p)(W+1) + pW(1-(2p)^m)}. \tag{4.6}$$

We can now express the probability τ that a station transmits in a randomly chosen time slot. As any transmission occurs when the backoff time counter is equal to zero, regardless of the backoff stage, it is

$$\tau = \sum_{i=0}^{m} b_{i,0} = \frac{b_{0,0}}{1-p} = \frac{2(1-2p)}{(1-2p)(W+1) + pW(1-(2p)^m)}. \tag{4.7}$$

As we can see, τ depends on the conditional collision probability p, which is still unknown. To find the value of p it is sufficient to note that the probability p that a transmitted packet encounters a collision is the probability that, in a time slot, at least one of the remaining $n-1$ stations transmit. Thus, we have

$$p = 1 - (1-\tau)^{n-1}. \tag{4.8}$$

Equations (4.7) and (4.8) represent a nonlinear system in the two unknowns τ and p , which can be solved using numerical techniques. It is easy to prove that this system has a unique solution. In fact, inverting (4.8), we obtain $\tau^*(p) = 1-(1-p)^{\frac{1}{n-1}}$. This is a continuous and monotone increasing function in the range $p \in (0,1)$, that starts from $\tau^*(0) = 0$ and grows up to $\tau^*(1) = 1$. Equation $\tau(p)$ defined by (4.7) is also a continuous and monotone decreasing function that starts from $\tau(0) = \frac{2}{(W+1)}$ and reduces down to $\tau(1) = \frac{2}{(1+2^m W)}$. Uniqueness of the solution is now proven noting that $\tau(0) > \tau^*(0)$ and $\tau(1) < \tau^*(1)$.

Throughput

To compute the throughput, we first analyze what can happen in a randomly chosen time slot. Let P_{tr} be the probability that there is at least one transmission

in the considered time slot. Since n stations contend on the channel, and each transmits with probability τ

$$P_{tr} = 1 - (1 - \tau)^n \tag{4.9}$$

The probability P_s that a transmission occurring on the channel is successful is given by the probability that exactly one station transmits on the channel, conditioned on the fact that at least one station transmits, i.e.,

$$P_s = \frac{n\tau(1-\tau)^n - 1}{P_{tr}} = \frac{n\tau(1-\tau)^n - 1}{1 - (1-\tau)^n} \tag{4.10}$$

Let S be the normalized system throughput, defined as the fraction of time the channel is used to successfully transmit payload bits. We are now able to express S as the ratio

$$S = \frac{E\left[\text{payload information transmitted in a time slot}\right]}{E\left[\text{length of a time slot}\right]}. \tag{4.11}$$

Being $E[P]$ the average packet payload size, the average amount of payload information successfully transmitted in a time slot is $P_{tr}P_sE[P]$, since a successful transmission occurs in a time slot with probability $P_{tr}P_s$. The average length of a time slot is readily obtained considering that, with probability $1 - P_{tr}$, the time slot is empty; with probability $P_{tr}P_s$ it contains a successful transmission, and with probability $P_{tr}(1 - P_s)$ it contains a collision. Hence, (4.11) becomes

$$S = \frac{P_sP_{tr}E[P]}{(1-P_{tr})\sigma + P_{tr}P_sT_s + P_{tr}(1-P_s)T_c}. \tag{4.12}$$

Here, T_s is the average time the channel is sensed busy (i.e., the slot time lasts) because of a successful transmission, and T_c is the average time the channel is sensed busy by each station during a collision. σ is the duration of an empty time slot. Of course, the values $E[P]$, T_s, T_c and σ must be expressed in the same unit.

Maximum Saturation Throughput

The analytical model given above is very convenient to determine the maximum achievable saturation throughput. Let us rearrange (4.12) to obtain

$$S = \frac{E[P]}{T_s - T_c + \frac{\sigma(1-P_{tr})/P_{tr} + T_c}{P_s}}. \tag{4.13}$$

As T_s, T_c, $E[P]$ and σ are constants, the throughput S is maximized when the following quantity is maximized:

$$\frac{P_s}{(1-P_{tr})/P_{tr}+T_c/\sigma} = \frac{n\tau(1-\tau)^{n-1}}{T_c^*-(1-\tau)^n(T_c^*-1)} \tag{4.14}$$

where $T_c^* = \frac{T_c}{\sigma}$ is the duration of a collision measured in time slot units σ. Taking the derivative of (4.14) with respect to τ , and imposing it equal to 0, we obtain, after some simplifications, the following equation:

$$(1-\tau)^n - T_c^*\{n\tau - [1-(1-\tau)^n]\} = 0 \tag{4.15}$$

Under the condition $\tau << 1$

$$(1-\tau)^n \approx 1 - n\tau + \frac{n(n-1)}{2}\tau^2$$

holds and yields the following approximate solution:

$$\tau \approx \frac{1}{n\sqrt{T_c^*/2}}. \tag{4.16}$$

Equation (4.15) and its approximate solution (4.16) are of fundamental theoretical importance. They allow us to explicitly compute the optimal transmission probability τ that each station should adopt in order to achieve maximum throughput performance within a considered network scenario (i.e., number of stations n).

The results of this model, as far as the throughput is concerned, can be viewed in Figure 4.4, where n represents the number of stations in the network.

It is obvious from Figure 4.4 that the maximum throughput is independent of the number of stations in the wireless network. On the other hand, we can see that in saturation conditions the behaviour of the protocol strongly depends on this number. Specifically, we can see that, as the number of the stations in the network increases, the throughput drops more sharply.

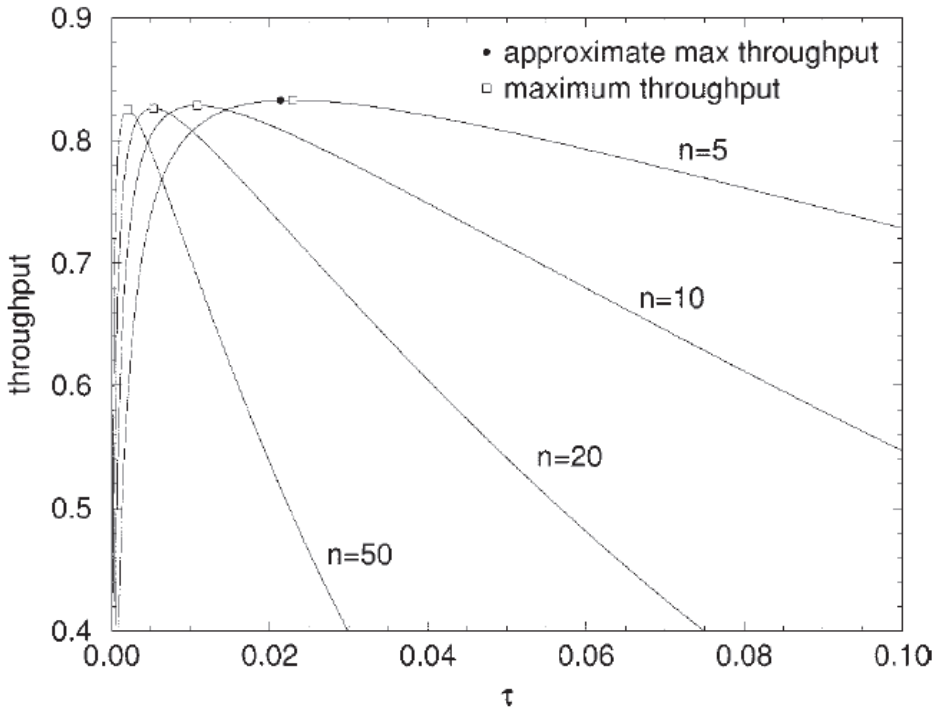

Figure 4.4: Throughput versus the transmission probability τ for the basic access scheme (Figure from [8])

4.3 Experimental results

In this section we present our experimental results and compare them to Bianchi's simulation and theoretical results of 802.11 DCF, as these were summarized in the previous sections. We tested our system with 2 and 4 users. In Figure 4.5, we plot our system's normalized throughput performance. The x-axis represents the total offered load of the network, which is, to our system, the equivalent of Bianchi's transmission probability τ.

As in the theoretical figure, we can also see here that the system's performance strongly depends on the number of the stations, especially in saturation conditions. Thus, we see that the system with 2 users has steadily higher throughput than the one with 4 users. This is something that we expect since, as the number of stations grows, we have more collisions in the network (especially when the stations are saturated) and consequently lower throughput. Furthermore, because of the small

number of stations we had at our disposal (up to 4) we could not see any significant drop in the saturation throughput. This is justified by the theoritical figure we showed above, where the saturation throughput for 5 stations drops very smoothly. As a result it would not be wise to compare our system's performance with Bianchi's simulation results in Figure 4.2, where the drop of the throughput was obvious but the number of stations was much bigger.

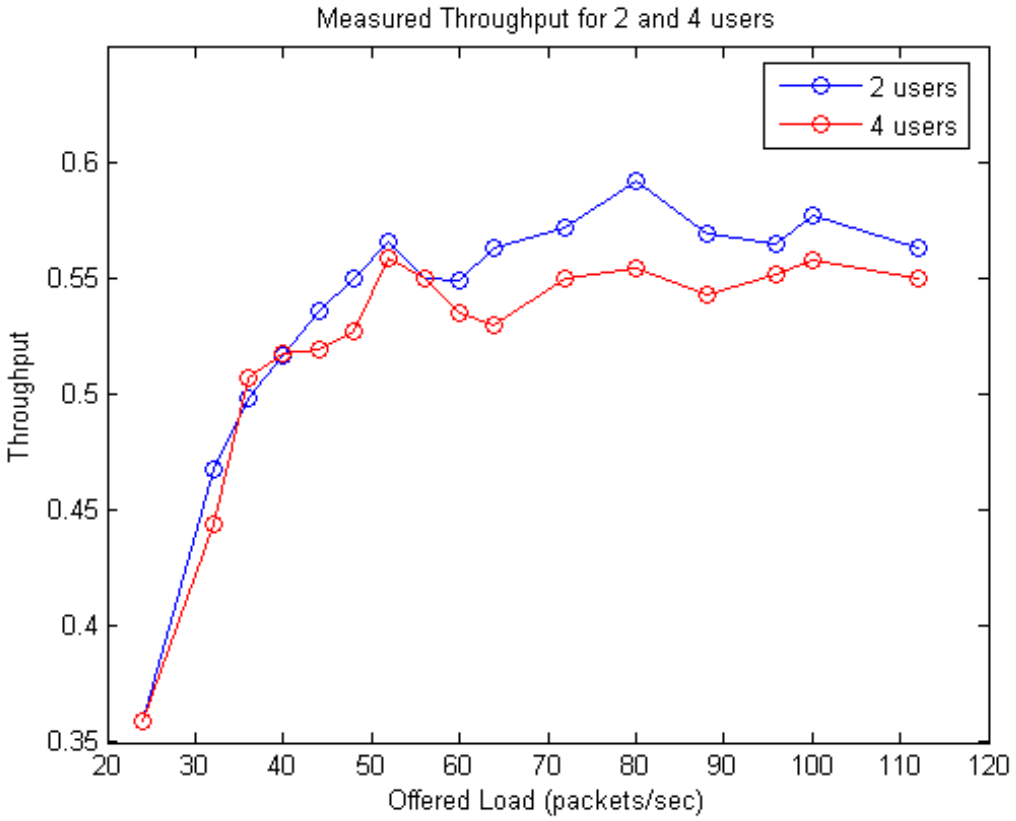

Figure 4.5: Experimental Throughput for 2 and 4 users

Bianchi's model also shows that the number of transmissions per packet significantly increases as the initial backoff window W reduces, and as the network size n increases. These results can be viewed in Figure 4.6 from [8].

We verified these results in our system as you can see in Figure 4.7. We note that in both theoretical and experimental results, the number of backoff stages m is the same for all the values of the initial size of the contention window.

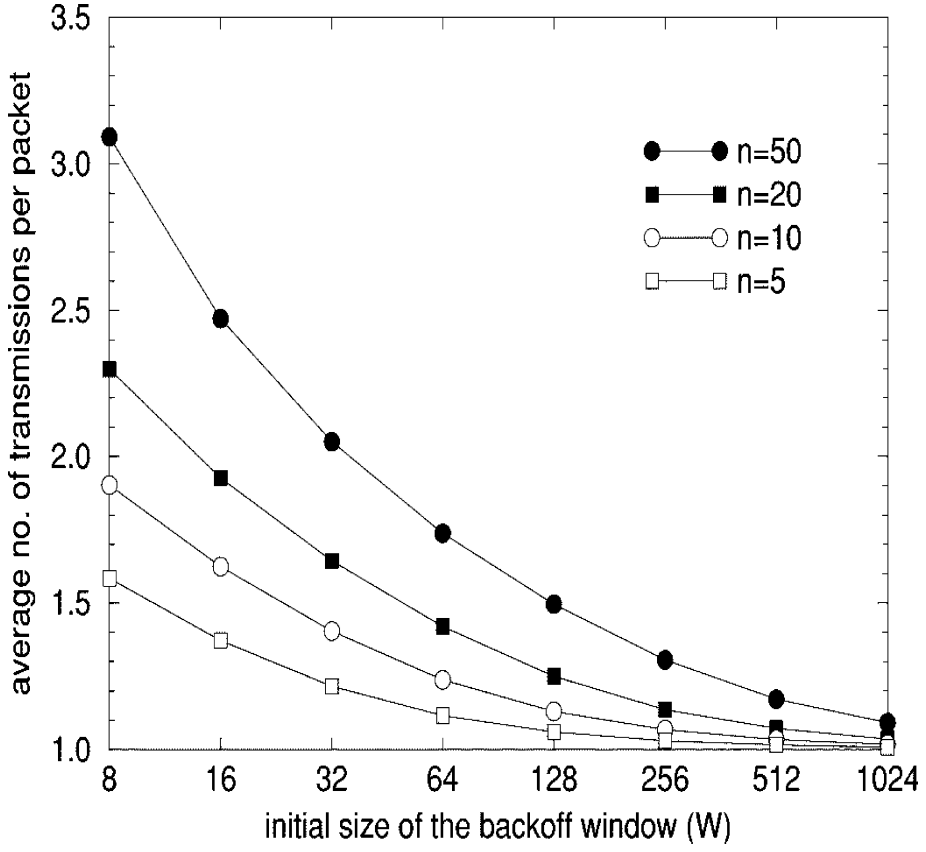

Figure 4.6: Average Number of transmissions per packet according to Bianchi's Model (Figure from [8])

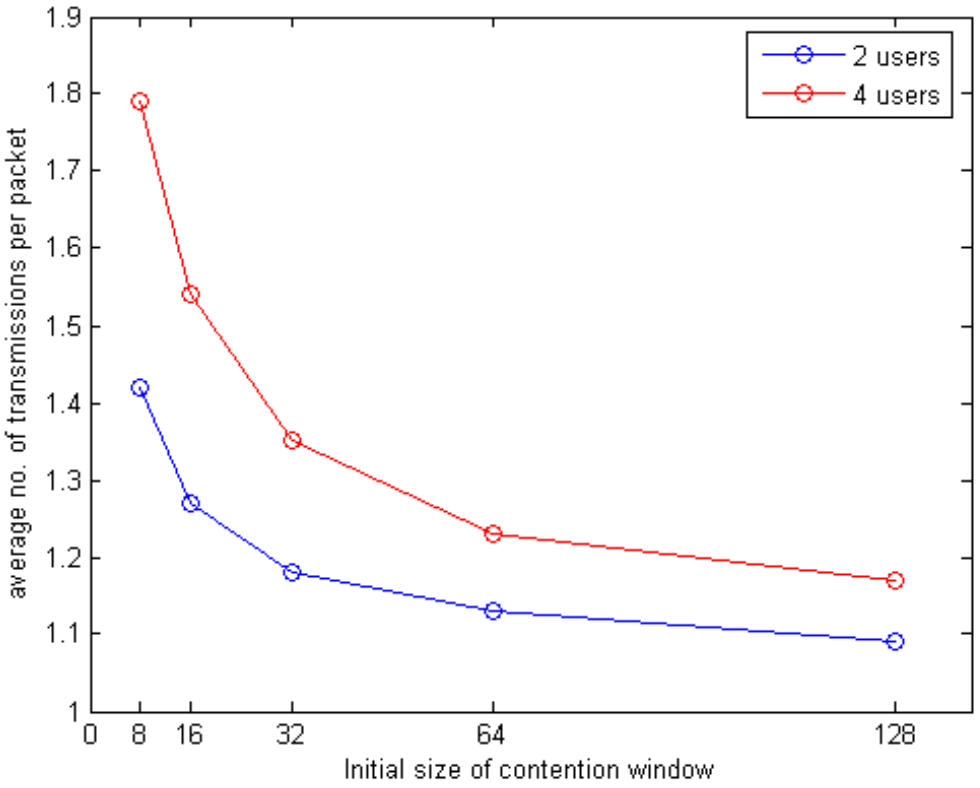

Figure 4.7: Average Number of transmissions per packet according to our system experimental results

Chapter 5

Future Work

In a future version of this SDR system, it would be useful to improve its performance in order to get to large data rates comparable to those expected, according to the IEEE standard for the 802.11 MAC protocol [11]. In order to accomplish this goal, we should reduce the delays associated with packet scheduling and processing as well as the Rx-Tx, Tx-Rx Turnaround times. To this direction, the authors in [10] suggest a minimum set of core MAC functions that must be implemented close to the radio (hardware) in high-latency SDR architectures, like ours, to enable high performance and efficient MAC implementations. These functions include: precise scheduling in time, carrier sense, backoff, packet recognition and access to physical layer information. Then, they define a split-functionality architecture that allows the functions to be implemented near the radio (hardware), while maintaining control on the host CPU through an API.

For performance reasons, it would also be important to use a PHY which could support much larger packets than those we used in our system. In this case we would also need a packetizer which can support different packet lengths during runtime, so that we can have different lengths for the packets and the Acknowledgments or other control messages.

Based on this SDR we can also experiment with different sophisticated PHY techniques, like OFDM or MIMO systems, and evaluate the achieved performance of these techniques in combination with our MAC protocol implementation. Furthermore, taking advantage of our cross layer interactions we could use a rate adap-

tive MAC protocol like the one implemented in [4]. This protocol uses the control messages of the DCF mode of IEEE 802.11 to perform opportunistic link level rate-adaptation. The goal is to use a higher rate when the wireless channel permits (i.e, the estimated link quality between the sender and the receiver is good enough).

Bibliography

[1] http://gnuradio.org/redmine/wiki/gnuradio/UsrpFAQIntro.

[2] M.Matigakis, "GNU Radio and USRP usage tutorial".

[3] http://gnuradio.org/redmine/wiki/gnuradio.

[4] K. Mandke, S.H. Choi, G.Kim, R. Grant, R.C. Daniels, W.Kim, R.W. Heath, Jr., and S.M. Nettles, "Early Results on Hydra: A Flexible MAC/PHY Multihop Testbed".

[5] E. Kohler, R. Morris, B. Chen, J. Jannoti and M.F. Kaashoek, "The Click Modular Router".

[6] R. Dhar, P.Steenkiste, "Supporting Integrated MAC and PHY Software Development for the USRP SDR".

[7] http://www.linuxhowtos.org/C_C++/socket.htm.

[8] G. Bianchi, "Performance Analysis of the IEEE 802.11 Distributed Coordination Function".

[9] Cisco Systems, "End-to-end QoS network design".

[10] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, P. Steenkiste Carnegie Mellon University, "Enabling MAC Protocol Implementations on Software-Defined Radios".

[11] ANSI/IEEE Std 802.11, 1999 Edition, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.