# Design and Implementation of a MAC protocol for Wireless Distributed Computing

Soumava Bera

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

In

Electrical Engineering

June 20, 2011

Blacksburg, VA

Keywords: Software Defined Radio, wireless distributed computing, MAC, OSSIE

Reliable data transfer, rate adaptation, Polling, CSMA/CA

# Design and Implementation of a MAC protocol for Wireless Distributed Computing

Soumava Bera

ABSTRACT

*The idea of wireless distributed computing (WDC) is rapidly gaining recognition owing to its promising potential in military, public safety and commercial applications. This concept basically entails distributing a computationally intensive task that one radio device is assigned, among its neighboring peer radio devices. The added processing power of multiple radios can be harnessed to significantly reduce the time consumed in obtaining the results of the original complex task. Since the idea of wireless distributed computing depends on a radio device forming a network with its peers, it is imperative and necessary to have a medium access control (MAC) protocol for such networks which is capable of scheduling channel access by multiple radios in the network, ensuring reliable data transfer, incorporating rate adaptation as well as handling link failures. The thesis presented here elaborates the design and implementation of such a MAC protocol for WDC employed in a practical network of radio devices configurable through software. It also brings to light the design and implementation constraints and challenges faced in this endeavor and puts forward viable solutions.*

*This thesis is dedicated to my father, mother and sister.*

# Acknowledgements

I would like to take this opportunity to express my sincere gratitude to Dr. Jeffrey H. Reed and Dr. Carl B. Dietrich for believing in me and giving me an opportunity to pursue my research interest. Your mentorship has constantly inspired me to strive for nothing short of excellence and your wealth of experience has made me wiser by the day.

I would also like to thank Dr. S. M. Hasan for introducing me to the area of wireless distributed computing. Your role has been pivotal in guiding me find my niche and follow through on it. I would like to thank Dr. Joseph Gaeddert and Tom Tsou from the bottom of my heart for their insightful and invaluable help whenever I needed. The knowledge and acumen that I have gained from you has been cardinal in my work. I would also like to thank Dr. Haris Volos and Dinesh Datla for motivating and mentoring me through my ups and downs. I would like to thank all my peers from the MPRG lab and office staff for cultivating an environment where I could learn by asking questions and making mistakes. Last but not least, I would like to thank my dear friends for their moral support through dispirited times in my life.

The last two wonderful years of my life at Virginia Tech have molded my perspective on life and have made me a better engineer. I will always treasure this unparalleled experience.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

In this fast paced evolution of innovations, new technologies emerge to surmount challenging problems and open the door to useful applications that can transform the world around us. Wireless distributed computing (WDC) is an emerging technology which can enable many applications which otherwise will remain unchallenged in today's world where our demands often surpass the capability of devices around us. The premise of wireless distributed computing requires a radio device to form at least a single hop network with peer radio devices to distribute and compute an intensive task. This renders salient advantages such as reduced per-node computational latency, energy and power consumption. Such unique purpose of WDC requires a MAC protocol which not only schedules transmissions among the radio devices efficaciously but also governs data distribution among peer radio devices and data retrieval from them. Traditional commercial multi-hop networks based on the fundamental OSI network model have a limited role defined in the MAC layer as the subsequent upper layers provide different additional functions. This hierarchy of functionality introduces a redundant layer traversal and processing delay potentially substantial to WDC applications. So it is desirable to enable the MAC protocol for WDC to bear additional functionalities like reliable data transfer, rate adaptation and link failure recovery. There is no such existing MAC protocol for WDC.

Existing commercial protocols (such as IEEE 802.11[1], etc.) are not suitable to the context of WDC networks mainly due to the following reasons:

- Commercial protocols use standardized and fixed parameters (example- bandwidth, operational frequency, slot times, etc.) which may not be supported by different radios due to hardware limitations (signal processing). For example, the channel sensing time of 50μs used in 802.11b cannot be achieved using USRP1[2] because of hardware constraints.
- The defined WDC architecture requires a certain degree of freedom to be able to reconfigure certain properties in the MAC layer and Physical layer to adjudicate efficient resource usage[3].
- The purpose of WDC is to distribute an intensive task among radio devices and retrieve outputs from them. This is governed by the MAC protocol. Traditional commercial networks do not serve this purpose even though they involve data sharing.  It must be noted that WDC is a different paradigm compared to wireless sensor networks. Sensor networks are multi-hop networks consisting of hundreds or thousands of nodes each with very limited memory and computation resources, typically used for monitoring and control. WDC networks have a different fundamental application .WDC networks are typically single hop networks with much lesser number of nodes compared to sensor networks where each node has a fair computational resources. A number of MAC protocols for sensor networks discussed in [4] are not suitable for WDC mainly because of the following main reasons:
  i) Objective of WDC network is functionally and fundamentally different.
  ii) WDC networks typically have star topology.
  iii) Nodes in a WDC network can be heterogeneous.

## 1.2 Goals

The overall goal of this work is to design and implement a reconfigurable MAC protocol for WDC networks on software defined radios. This will serve as a benchmark MAC protocol in the realm of WDC and its working will be demonstrated on a prototype network of radio devices. Implementation of the MAC protocol will need the following:

i) A platform to develop the protocol.

ii) A MAC framework to define the frame structure for different communication messages.
iii) A central algorithm which integrates all the functionalities of the MAC protocol.

The design is implemented using a software package OSSIE and the radio frontend used is an open source hardware - USRP1[2, 5]. OSSIE is natively suited for streaming data applications (like FM). The key challenge in implementing a MAC protocol on OSSIE is to extend its capability to support packet data over a network of radio devices for which a special interface between OSSIE and USB is employed. This work will also explore all the relevant design and implementation constraints and challenges and present viable solutions. The components and waveforms created on OSSIE provide the background infrastructure for the MAC protocol.

This work will explain the design of the MAC framework which makes it feasible to define the different types of communication messages that implement the protocol. Certain latency and infrastructure issues impose practical constraints on the system. This work will explain implementation of feasible channel access protocols and scheduling mechanisms in this regard. The integration of different aspects of this MAC protocol such as scheduling, reliable data transfer, rate adaptation and timer structure is also addressed. The flexible nature of this framework enables it to function on a wide range of radios with different hardware (ADC, DAC, etc.) limitations. The outcome of this work will provide a platform to further research and development on MAC on software defined radios.

## 1.3 Accomplishments

The primary accomplishments of this thesis are:

i. Support for packet based data on OSSIE for use in a network of radios.
ii. Design of waveform and components on OSSIE that make the MAC protocol feasible.
iii. Reconfigurable MAC framework that can also meet the limitations of different radio device hardware.

iv. Design and implementation of channel access and message scheduling schemes that mitigate latency constraints imposed by the system.
v. A rate adaptation algorithm that befits the protocol design.
vi. A working MAC protocol for WDC which incorporates reliable data transfer, rate adaptation and link failure recovery.

i. & ii provide the platform to develop the MAC protocol, iii provides the MAC framework that defines format of communication and iv-vi help realize the different functionalities within the MAC protocol and how they integrate with each other. These accomplishments yield in a fully functional MAC protocol that can be demonstrated practically on a network of radios.

## 1.4 Organization

The work presented in this thesis is organized into the following chapters:

- *Chapter 2* provides the relevant background information needed to understand the purpose, principle and working of the MAC protocol.
- *Chapter 3* discusses the infrastructure designed for the MAC protocol which includes OSSIE waveforms and components. It throws light upon the several factors considered in the design and their significance.
- *Chapter 4* describes the MAC framework which entails the frame structure and the various types of messages exchanged in the MAC protocol. It gives an idea of how the events in the MAC protocol are classified and delineates the sequence events taking place throughout the intercommunication among nodes in a WDC application.

- *Chapter 5* explains the significance and working of the channel access scheme and message scheduling scheme employed on the downlink aspect of the MAC protocol. It elaborates the implementation of the various downlink events at both master and slave nodes and also explains how reliable data transfer, rate adaptation and link failure recovery are incorporated.

- *Chapter 6* talks about two alternative uplink protocols involving two different multiple access schemes. It elucidates the significance and implementation of these protocols, design constraints and viable solutions.
- *Chapter 7* describes the implementation of Hybrid ARQ in the context of reliable communication and design of a rate adaptation algorithm that enables a variation of Hybrid ARQ.
- *Chapter 8* presents the results of practical tests performed on different aspects of the MAC protocol. It provides an analysis of the performance observed during data transfer and rate adaptation based on a defined set of metrics. This chapter discusses the latency measurement procedures and brings to light the latency constraints in the system. It also identifies the various factors affecting data rate and their relationships.
- *Chapter 9* provides the conclusion of this work and given directions for future work.

# Chapter 2 Background information

## 2.1 Wireless Distributed computing (WDC)

The idea of WDC entails multiple independent radio devices collaborating over the air to form a (WDC) network to perform a complex task. This yields a significant increase in performance and efficiency over a single radio device. WDC can serve its purpose in wireless sensor networks (WSNs), tactical radios, and commercial smart phones. Applications that benefit from WDC involve complex communication and computation tasks that impose stringent quality of service (QoS) constraints, (power/energy consumption, latency, and range) as well as stringent resource constraints (computational power). It is not feasible for a single radio device to meet these requirements but if multiple nodes form a WDC network and distribute the complex task among them these constraints can be overcome. Studies have shown that WDC has considerable benefits, particularly when the computational cost dominates over the communication overhead, which is the case when complex computational tasks are distributed among multiple network devices in short-range networks [6-8].

In WDC, when a radio obtains a complex and computationally intensive task, it forms a WDC network with the peer radio devices. In this work the network is considered to have a star topology with a central master node and peripheral slave nodes as shown in Figure 2.1 below. The radio device which has the original task takes on the role of the master node while the other radio devices become the slave nodes. The master node determines the number of available slave nodes, divides the original task into smaller sub-tasks using resource allocation algorithm/s and sends them to the respective slave nodes for processing. Each slave node finishes its share of processing and sends a simplified result to the master node. The master node uses the outputs from all the slave nodes to yield the final result [6-8]. It has been shown in [3, 9] that for a sufficiently complex task, WDC applied over single hop networks provides benefits in power consumption.

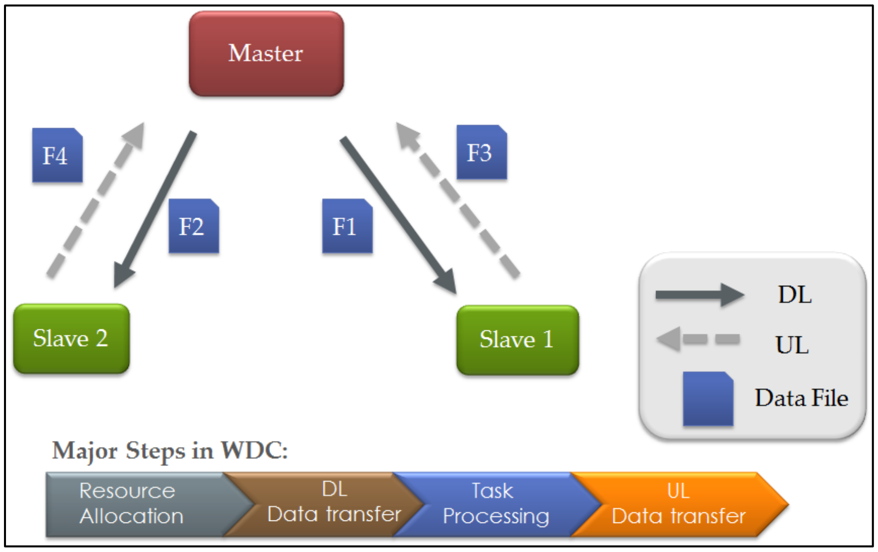


Figure 2.1: WDC network

## 2.2 Software defined radio

Software defined radio (SDR) refers to a communication system where components (such as filters, mixers, modulators/demodulators, amplifiers, etc.) typically implemented in hardware (in a hardware radio) are implemented in software on a personal computer or an embedded computing device. The major advantages of software defined radio are flexibility and cost effectiveness. A typical software defined radio consists of a general purpose processor (GPP) running a specialized software package interfaced with a flexible signal processing hardware and radio front end. Significant amounts of signal processing are typically handled by the general-purpose processor, rather than the specialized hardware [10].

A conventional (hardware) radio relies on specific hardware components for communication. But SDR provides flexibility in terms of reuse of components on multiple platforms and using same platform for multiple SDR based communication systems. It allows to dynamically configure a radio which can receive and transmit widely different radio protocols (sometimes referred to as a waveforms) based solely on the software used. Software radios have significant utility for the military and cellular services, both of which must support the rapidly evolving newer standards with different requirements in throughput and bandwidth. SDR provides a very cost effective way to adopt different standards using the same hardware i.e. without any upgrade in hardware. In United States of America, for military applications, the Joint Tactical Radio System (JTRS) specification provides inter-operability among radios, used by warfighters and supports multiple waveforms from multiple defense contractors[10].

The software defined radio technology plays a key role in making wireless distributed computing possible. The MAC layer of WDC and the physical layer are implemented on the GPP. The physical layer configures the components on USRP[2] to facilitate transmission and reception. The flexibility of SDR also encourages continuous improvement in security functions to keep the radio functionality safe.

## 2.3 Network layers

The standard model for networking protocols and distributed applications is the International Standard Organization's Open System Interconnect (ISO/OSI) model[11]. It defines seven network layers. Every device in a network needs an organized way to communicate with other devices in the network. In order to make this feasible a set of rules and conventions for communication between network devices need to be defined. This constitutes a network protocol. Protocols operate at one or more layers in the network model.

Layers in the OSI model ordered from lowest level to highest are (1) physical (2) data link (3) network (4) transport (5) session (6) presentation and (7) application.

The OSI model makes network designs more extensible as new protocols and other network services are generally easier to add to a layered architecture than to a monolithic one. The Physical Layer of the OSI model is responsible for bit-level transmission between network nodes. The data link layer provides the functional and procedural means to transfer data between network entities and might provide the means to detect and possibly correct errors that may occur in the Physical Layer. The Network Layer of the OSI model is responsible for establishing routes for data transfer through the network. IP is one protocol that operates in this layer. It is necessary in a multi-hop network. The Transport Layer of the OSI model is responsible for delivering messages between networked hosts and explicit addressing of destinations. It is responsible for fragmentation and reassembly of data. TCP and UDP are commonly used transport layer protocols. The Session Layer of the OSI model is responsible for establishing process-to-process communications between networked hosts. The Presentation Layer of the OSI model is responsible for defining the syntax which two network hosts use to communicate. It also serves in encryption and compression. The Application Layer of the OSI model is responsible for providing end-user services, such as file transfers, electronic messaging, e-mail, virtual terminal access, and network management. This is the layer with which the user interacts[11].

By separating the network communications into logical smaller pieces, the OSI model simplifies how network protocols are designed. The OSI model was designed to ensure different types of equipment (such as network adapters, hubs, and routers) would all be compatible even if built by different manufacturers. A WDC network model is inspired by the OSI model but has certain distinctions.

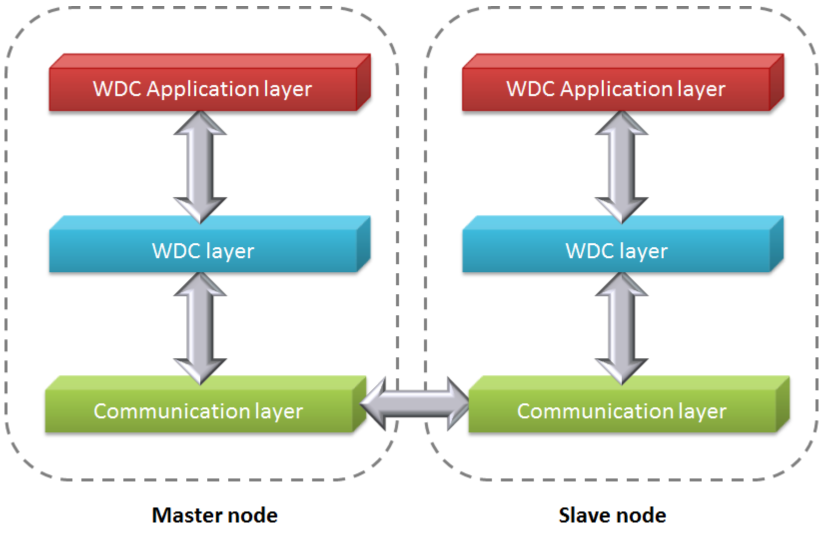


Figure 2.2: Network Layer Model for WDC

Figure 2.2 above shows a simplified network model for WDC. Each radio device or node in a WDC network has the three main layers: WDC application layer, WDC layer and communication layer. The WDC layer houses the WDC architecture which runs resource allocation algorithms and exercises controls over the lower layers. The communication layer comprises of the transport layer, network layer, physical layer and the medium access control layer. WDC is typically applicable to single hop networks where a network layer (protocol) is not necessary. At the data link layer, network devices communicate directly via a physical channel over a single hop, whereas at the transport layer, different logical channels are used by different participating applications at different end hosts over any number of hops and reliable data transfer is ensured over each such connection. It is quite intuitive that transport layer protocols are more suitable to multi-hop networks. In case of single hop networks, the transport layer would introduce redundant overhead as well as delay due to increased processing and traversal across the layers.

The medium access control layer is a major part of the data link layer and is the subject of focus in this thesis. While the physical layer provides means of bit synchronization, different modulation schemes for different data rates, signal processing and channel sensing, the MAC layer defines rules that specify how data is packaged, sent and received and enables the devices to access the channel, identify and make connections with each other. The MAC layer protocol for WDC is given additional functions such as reliable data transfer, rate adaptation and link failure recovery so that the WDC network can bypass redundant overhead and processing due to the transport layer. The unit of data at the MAC layer (data link layer) is termed as a frame while the unit of data at the transport layer is termed as a packet.

## 2.4 OSSIE

### 2.4.1 Structure

OSSIE or Open Source SCA Implementation Embedded project is an open source software development package designed for rapid prototyping and configuring software defined radios. The main objective of OSSIE is to further education and research in the field of wireless communications. The software package is based on the SCA (software communications architecture) core framework and provides a set of tools for rapid development of SDR components and waveforms applications. The SCA is an open architecture developed by the U.S. Department of Defense to assist in the development of SDRs. The SCA provides an implementation-independent set of rules to design SDRs which allow radio applications to be portable to a wide range of platforms (fixed communication hubs, handheld devices, etc.). What makes this possible is a standard operational environment (OE) which allows users to deploy and run waveforms across platforms with different hardware components, device drivers, or transport mechanisms. The OE is divided into three layers of abstraction: the operating system (OS), CORBA middleware (Common Object Resource Broker Architecture), and the core framework (CF). The OSSIE package is developed for the Linux operating system[12, 13].

An SCA component is basically a software object, which by itself is a standalone entity that performs some sort of signal processing or control functionality required by a waveform. SCA applications are made up of interconnected software components that run as separate processes. The connections between the components are handled by CORBA middleware. The components are deployed on executable devices and may be connected via standard interfaces with other devices (e.g., an RF front-end or sound card) or components. A component-based approach enables flexibility, modularity and reusability. Components can have different implementations for different devices (like sound card, radio front end device, etc.) and are connected to them through standard interfaces. A waveform is a set of transformations applied to information that is transmitted over the air and the corresponding set of transformations to convert received signals back to their information content. SCA waveforms are modular, distributed radio applications made of one or more components [12-14].

OSSIE Standard Interfaces is a class library interacts with the CORBA interfaces. The library facilitates code reuse for many common data types and makes it easy for the users/developers to work with CORBA. Currently supported IDL interfaces include basic real and complex data representations in 8, 16, and 32 bit sizes passed in the form of CORBA sequences[13, 14].

The middleware used by OSSIE is CORBA, a standard software defined by the Object Management Group (OMG). Its aim is to enable software modules written in a variety of different computer languages and running on different platforms to be operate together. CORBA makes it possible for the development of SCA waveforms independent of the actual deployment configuration on distributed modular platforms thereby allowing distribution of applications. It creates a flexible software bus to support modular, reconfigurable platforms creating a layer of abstraction between the applications and platform-specific elements such as real time operating system and transport layer. For a given application, different components can be deployed on different processors, boards, computers, or networks and yet appear as if they were located on the same platform [12, 13].

A Core Framework is a set of cooperating classes that make up a reusable design for a specific class of software. The SCA defines, in the CF, a set of interfaces that govern the deployment and management of waveforms and their components. These interfaces define low-level architectural details, allowing developers to focus on application design. The SCA CF interfaces are defined in CORBA's Interface Description Language (IDL) [12, 13].

The SCA CF includes:

- *Base application interfaces* provide a common mechanism for the control and configuration of software components. All waveform components are required to implement the base application interfaces[12].
- *Base device interfaces* allow interaction with physical hardware devices by providing a proxy to the rest of the framework. This abstraction allows non-CORBA-enabled elements to interact with other components and the rest of the framework[12].
- *Domain Profile* contains all the information regarding applications and platforms within the SCA. These files describe in Extensible Markup Language the interfaces, capacity models, properties, inter-dependencies, interconnections, and logical location of each and every component within the domain, system hardware as well as waveform software structure [12].

## 2.4.2 Development Environment

OSSIE waveform developer (OWD) provides a graphical user interface (GUI) that allows the developer to design new software components and interconnect existing components to create waveform applications. OWD allows the user to select from a library of available components to construct an SCA-based waveform or create the skeleton code for user defined C++ or Python components. While building a component, the number and datatype/s of the input/output ports need to be specified along with the datatype/s of the parameters (or knobs) if any, which will change the user defined properties of the components. OWD will generate the source code for the structure of the component while interacting with SCA framework and CORBA. The developer will have to define the signal processing functionality in C++ code within the user space of a component. The OSSIE Eclipse Feature (OEF) makes it possible to develop components and waveform applications, launch the domain manager and device managers, launch ALF, and, if desired, launch the legacy OWD, all from within the Eclipse GUI. ALF is waveform application visualization and debugging tool that allows a developer or user to launch waveforms on the target platforms, display them in block diagram form, and probe component ports using readymade or custom-developed plug-in tools such as a spectrum and constellation plotting tool. These tools have been developed using Python scripting language. The OSSIE Waveform Dashboard is a tool that provides an interactive GUI for installing and running waveforms, and configuring component properties [12-14].

### 2.4.3 Why OSSIE?

Gnuradio[15] is an alternative SDR development package. Some of the major benefits of using OSSIE are :

- OSSIE has a higher input data rate for the receiver path than Gnuradio[16].
- Component design is much simpler in OSSIE. Gnuradio code is very hierarchical and has a steep learning curve.
- The average delay in successive file read operations is lesser and more uniformly distributed in OSSIE compared to GNU Radio[16].
- The underlying SCA framework of OSSIE allows deployment of a single waveform over multiple devices[12].

## 2.5 USRP

The Universal Software Radio Peripheral or USRP enables a general purpose computer to function as software radio. The basic idea is that all of the waveform-specific processing, like modulation and demodulation are done on the host CPU while all of the high-speed general purpose operations like digital up and down conversion, decimation and interpolation are done on the FPGA of USRP.

This work has been demonstrated using USRP1s which enable engineers and designers to create a software defined radio on a low budget and with a minimum effort. Before proceeding to the later chapters, it is important to acquire an understanding of what happens to the data at the USRP. A section of USRP1 representing the major component blocks are shown in Figure 2.3.

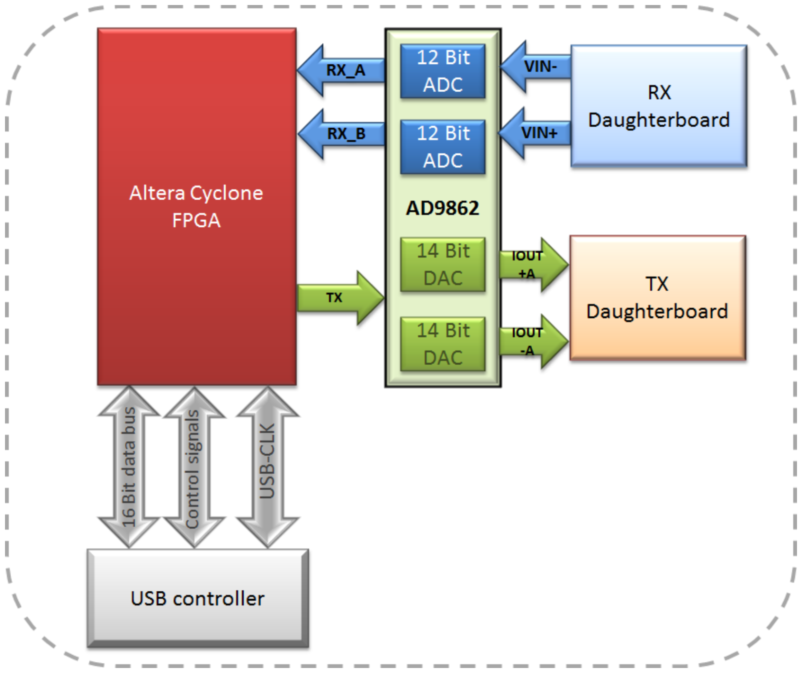


Figure 2.3: USRP Block Diagram

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, connects to a USB2 interface chip, the

Cypress FX2 (programmable High-Speed USB controller), and on to the computer. The USRP connects to the computer via a high speed USB2[5].

When the USRP is plugged in to the USB for the first time, the host-side (laptop) library downloads the (8051 microcontroller) code that defines the behavior of the USB peripheral controller. When this code boots, the host sees the device connected and the USRP firmware starts running and defines the USB endpoints, interfaces and command handlers. It commands to load the FPGA. The library code on the host side downloads the FPGA configuration bit stream[5].

The standard FPGA configuration includes digital down converters (DDC) implemented with 4 stages cascaded integrator-comb (CIC) filters. CIC filters are very high-performance filters using only adds and delays. In the RX path, we have 4 ADCs (Analog to digital converter), and 4 DDCs (Digital to analog converter). Each DDC has two inputs I and Q. It down converts the signal from the IF band to the base band and then decimates the signal so that the data rate can be adapted by the USB 2.0 and is suitable for the computer's computing capability. All samples sent over the USB interface are in 16-bit signed integers in IQ format, i.e. 16-bit I and 16-bit Q data (complex) which means 4 bytes per complex sample. Even though USB2 supports a raw signaling rate of 480 Mbps, USRP1 can support 256 Mbps (32 megabytes/s). It is not possible to achieve the full 480 because there is overhead from packet headers, time between packets, etc. This means we can have a maximum of (32MBps /4 Bytes) 8 Mega complex samples per second across the USB 2.0. The story is pretty much same on the transmitter side. A baseband I/Q complex signal to the USRP is interpolated by the digital up converter (DUC), up-converted to the IF band and finally sent through the DAC [5]. The daughterboards used in the practical setup are RFX 400 series which support frequency range of 400MHz - 500MHz[2].

In Figure 2.3, a TX daughterboard has a pair of differential analog current outputs (IOUT +A/IOUT -A) which are updated at 128 MS/s. An RX daughterboard has 2 differential analog inputs (VIN-/VIN+) which are sampled at a rate of 64 MS/s [5].

# Chapter 3

# Implementation on OSSIE

## 3.1 Waveform Overview

This chapter details the platform on which the MAC protocol has been developed. It sheds light on the infrastructure needed for the functioning of the MAC protocol which enables interaction between the physical layer and the MAC layer.

The MAC protocol for wireless distributed computing has been implemented on OSSIE[17] in a waveform application. The waveform consists of multiple components each defined with segregated functionality to facilitate modularity. The MAC protocol is housed in one generic component on a particular node so that it can interface with different physical layer libraries in different waveforms. The structure of the waveform was intended to be simple avoiding any feedback loops due to complexities in thread handling. The waveform adopts the WDC network model to achieve a linear data flow.

Data flows from the link layer to the physical layer at the transmitter and from physical to link layer at the receiver. The physical layer and the MAC layer are both implemented on the GPP. The hardware RF front end simply transmits or receives the physical layer information. The role of RF front end is served by USRP (URSP1) [2]. The USRP1 is connected to the GPP (of a portable device, like laptop) through a USB 2.0 connection.

The following figures represent screenshots of the waveforms run on the master and the slave nodes respectively.

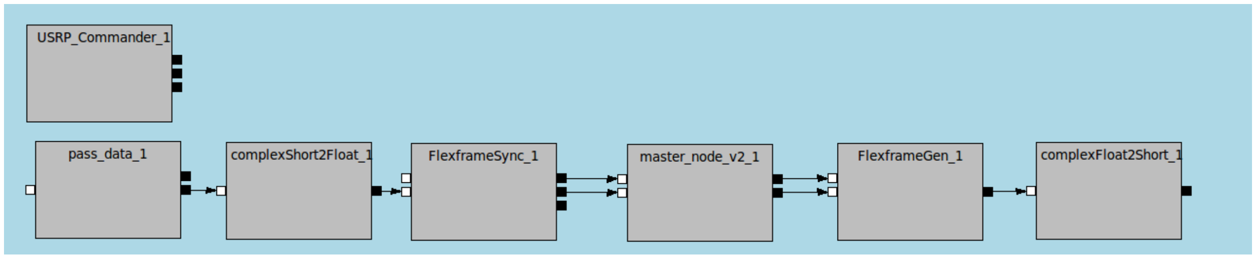


Figure 3.1: OSSIE waveform at the Master node

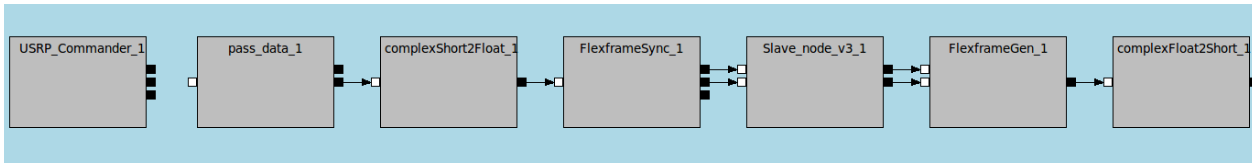


Figure 3.2: OSSIE waveform at a Slave node

In Figures 3.1 and 3.2, each grey box represents a component. The dark dots represent ports of a particular standard interface. The readymade components provided by OSSIE are *pass_data* and *USRP_Commander*. All the other components have been added to the component list on OSSIE. The components *complexShort2float*, *FlexFrameSync*, *FlexFrameGen*, and *complexFloat2Short* represent the physical layer processing while the (MAC) components *master_node_v2* and *Slave_node_v3* house the entire MAC layer processing. The physical layer components, *FlexframeSync* and *FlexframeGen* act as interface to the 'liquid' radio libraries, which are used in the physical layer processing of data. Each component has its separate process. The structure of the OSSIE waveform is similar on both the master and slave nodes with the key difference in one component which houses the implementation of the MAC protocol.

The following figure illustrates the port types of the interfaces and the relevance to the layered network model discussed previously.

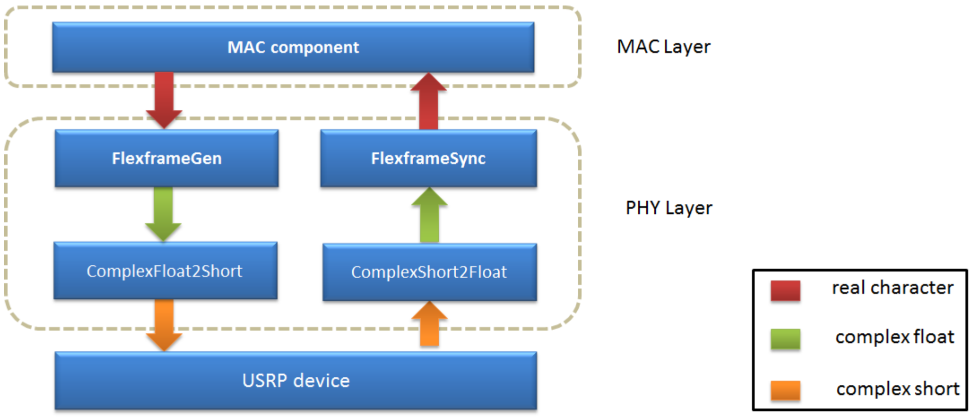
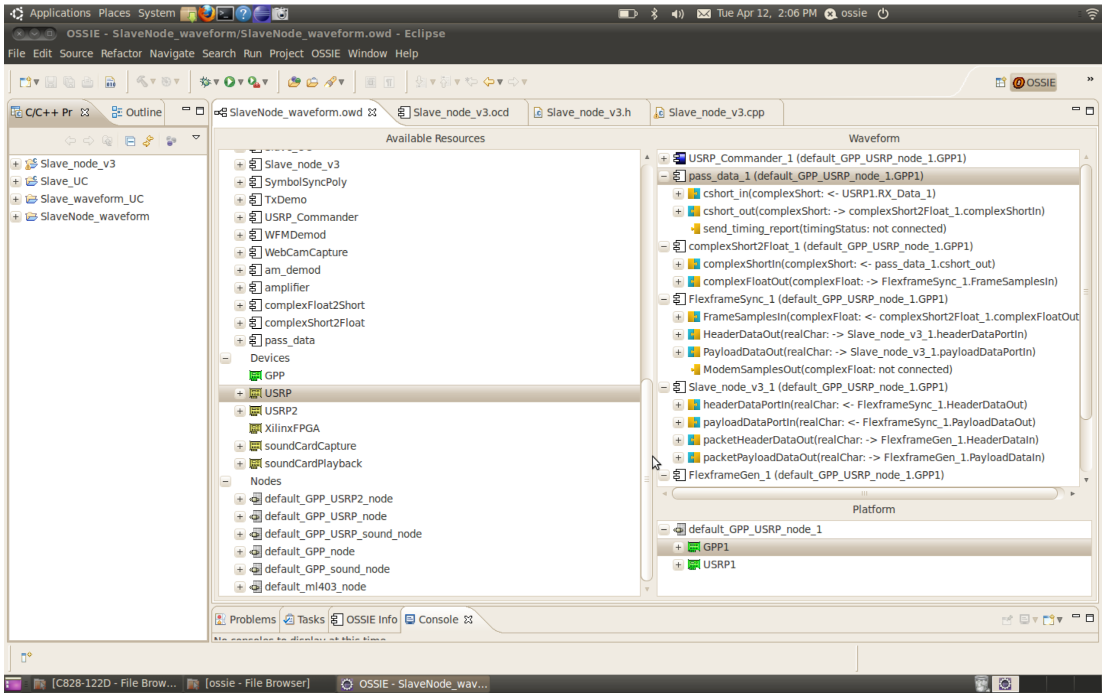


Figure 3.3: Relation between waveform and network layer model

When data from the upper layers is passed on to the MAC layer, it assembles them into a 'first in first out' buffer or a file, from which it prepares data frames till there is no more data left to be read.

The following figure shows the aforementioned waveform on a slave node being constructed using Eclipse (Waveform design platform).


Applications Places System
Tue Apr 12, 2:06 PM ossie
OSSIE - SlaveNode_waveform/SlaveNode_waveform.owd - Eclipse
File Edit Source Refactor Navigate Search Run Project OSSIE Window Help
OSSIE
C/C++ Pr
Outline
SlaveNode_waveform.owd
Slave_node_v3.ocd
Slave_node_v3.h
Slave_node_v3.cpp
Available Resources
Waveform
Platform
Problems Tasks OSSIE Info Console
[C828-122D - File Brow...
[ossie - File Browser]
OSSIE - SlaveNode_wav...

```
Slave_node_v3
Slave_UC
Slave_waveform_UC
SlaveNode_waveform
```

```
Available Resources
  Slave_node_v3
  SymbolSyncPoly
  TxDemo
  USRP_Commander
  WFMDemod
  WebCamCapture
  am_demod
  amplifier
  complexFloat2Short
  complexShort2Float
  pass_data
Devices
  GPP
  USRP
  USRP2
  XilinxFPGA
  soundCardCapture
  soundCardPlayback
Nodes
  default_GPP_USRP2_node
  default_GPP_USRP_node
  default_GPP_USRP_sound_node
  default_GPP_node
  default_GPP_sound_node
  default_ml403_node
```

```
Waveform
USRP_Commander_1 (default_GPP_USRP_node_1.GPP1)
pass_data_1 (default_GPP_USRP_node_1.GPP1)
  cshort_in(complexShort: <- USRP1.RX_Data_1)
  cshort_out(complexShort: -> complexShort2Float_1.complexShortIn)
  send_timing_report(timingStatus: not connected)
complexShort2Float_1 (default_GPP_USRP_node_1.GPP1)
  complexShortIn(complexShort: <- pass_data_1.cshort_out)
  complexFloatOut(complexFloat: -> FlexframeSync_1.FrameSamplesIn)
FlexframeSync_1 (default_GPP_USRP_node_1.GPP1)
  FrameSamplesIn(complexFloat: <- complexShort2Float_1.complexFloatOut
  HeaderDataOut(realChar: -> Slave_node_v3_1.headerDataPortIn)
  PayloadDataOut(realChar: -> Slave_node_v3_1.payloadDataPortIn)
  ModemSamplesOut(complexFloat: not connected)
Slave_node_v3_1 (default_GPP_USRP_node_1.GPP1)
  headerDataPortIn(realChar: <- FlexframeSync_1.HeaderDataOut)
  payloadDataPortIn(realChar: <- FlexframeSync_1.PayloadDataOut)
  packetHeaderDataOut(realChar: -> FlexframeGen_1.HeaderDataIn)
  packetPayloadDataOut(realChar: -> FlexframeGen_1.PayloadDataIn)
FlexframeGen_1 (default_GPP_USRP_node_1.GPP1)

Platform
default_GPP_USRP_node_1
  GPP1
  USRP1
```

Figure 3.4: Waveform development using Eclipse

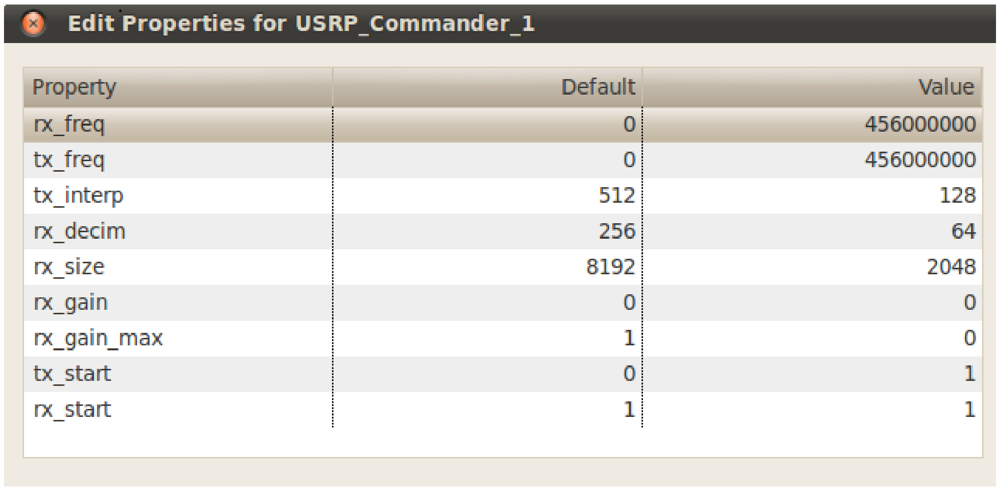
In Figure 3.4, the available resources show the available components, devices and nodes. A node basically represents the different hardware devices that will be interacting within a waveform. The waveform panel shows how different ports of different components are connected via compatible standard interfaces. In this case, it is a waveform on the slave node side. The platform panel shows the node/s the waveform is using: Host side GPP and the USRP device.

## 3.2 Components

This section contains brief description of each component and highlights necessary considerations for design.

### 3.2.1 Description

- ***USRP_Commander***: It is axiomatic that we need an interface between OSSIE on the host GPP and the hardware. This will be the first component on the receive path and the last component on the transmit path. The USRP commander serves this function in addition to configuring the transmitter/receiver carrier frequency, RX buffer size and USRP interpolation and decimation rates and checking for daughterboards and assigning the sub-device. It helps in pushing the data samples to the USB controller. The samples pushed from the USRP commander arrive at the FPGA through the USB bus. Section 2.5 provides an insight about what happens at the USRP. The following figure shows the configuration used in *USRP_Commander*.


**Edit Properties for USRP_Commander_1**

| Property | Default | Value |
|---|---|---|
| rx_freq | 0 | 456000000 |
| tx_freq | 0 | 456000000 |
| tx_interp | 512 | 128 |
| rx_decim | 256 | 64 |
| rx_size | 8192 | 2048 |
| rx_gain | 0 | 0 |
| rx_gain_max | 1 | 0 |
| tx_start | 0 | 1 |
| rx_start | 1 | 1 |

Figure 3.5: Basic waveform settings

- ***Pass_data***: This component simply relays the received samples from the USRP to the next component.

Components present exclusively on the RX path:

- ***ComplexShort2Float***: the data received from the USRP is in the format of complex short I-Q samples. This component converts the complex short (fixed point) into float I-Q samples for floating point arithmetic in the physical layer signal processing. The *Flexframe* components use floating point data due to its greater accuracy and performance. [18] discusses the advantages of floating point data over fixed point data. This component also has a knob of adjustable gain.

- ***FlexframeSync***: This component calls the underlying Liquid radio library which detects the presence of a frame, corrects for gain, carrier, and sample timing offsets (channel impairments) in the complex baseband samples, decodes the data and passes it to the next component. This component sends to the next connected component the MAC header and payload through two different output ports of the standard interface type: real character sequence.

Components present exclusively on the TX path:

- ***complexShort2float***: Since the USRP accepts data in the form of complex short samples. This component converts the float I-Q samples to short I-Q samples for fixed point operations in the FPGA. This component also has a knob for adjustable gain.
- ***FlexframeGen***: This component modulates the data bits and then interpolates these symbols with a matched filter to produce a frame at complex baseband. This component accepts from the preceding connected component the MAC header and payload through two different input ports of the standard interface type: real character sequence.

MAC components:

The *master_node_v2* and *slave_node_v3* components participate in both the TX and the RX path of data flow. These components work with the *Flexframe* components to facilitate MAC layer protocol for WDC. The original *Flexframe* components developed to interface Liquid radio with OSSIE have been updated with certain additional functions to realize the MAC protocol. Each of the MAC components have two parallel threads one for the transmitter chain and other for the receiver chain. Each such thread consists of a continuously running processing loop. The transmitter thread handles the output ports of the component while the receiver thread handles the input ports of the component. The type of data that flows in and out of these ports is a sequence of real characters. The threads communicate with each other through shared variables or flags in the user space. The processing loop further consists of sections of code, each identified by a combination of flags or indicator variables.

## 3.2.2 Design considerations

### 3.2.2.1 Continuous flow of data

In Figures 3.1 and 3.2, we can see that data flows from *FlexframeSync* component to *master_node_v2* or *Slave_node_v3* component. The data that arrives at these MAC components essentially comprises of the MAC header and payload. Each component acquires data from its input ports or sends data through its output ports within a continuous processing loop. A continuous flow of data is ensured throughout a waveform as long as the processing loop in each member component of the waveform is actively running. If one component doesn't receive data, the component simply waits for the data and the loop doesn't move forward to the next iteration. A component pushes data to the next connected component through its output ports. Packet data arrives in bursts which causes an idle time throughout the waveform. The MAC components are the innermost components in the receive path. If the FlexframeSync component doesn't find a valid physical layer header it does not detect a valid frame. This information needs to be relayed to the subsequent MAC component so that the wait timers in the designed MAC protocol can be managed effectively.

In order to resolve this problem, void frames are used. Whenever the *FlexframeSync* component detects the channel as idle, it pushes an empty MAC header and a null payload (i.e. Payload length of size zero) to the next connected component. An empty header consists of a known pattern. As soon as the subsequent MAC component receives this empty header and null payload, it recognizes this pattern as a void frame thereby knowing that the channel is idle. In this case, the pattern is defined as seven consecutive zeros in the eight byte header. Now the question that arises is that how the *FlexframeSync* component knows when to send the next connected component a void frame. Before pushing a void frame, the *FlexframeSync* component generally waits for a defined number of input samples during which no recognizable header has been received. The larger is this number of samples, the longer is the wait time i.e. more is the delay experienced by the subsequent component to know about an idle channel. However, if the wait time is really small, it will keep the GPP of the host quite busy. In our experimental setup, we used laptops with a large enough computing resources, so the number of samples is kept small (defined as five). We now see that by sending void frames from the *FlexframeSync* component to a MAC component, the processing loops in the latter are kept running even when a node doesn't receive any message.

#### 3.2.2.2 Channel status

An integral part of CSMA/CA (Carrier sense multiple access with collision avoidance) protocol is carrier sensing. To be able to perform carrier sensing, the MAC component should be able to as soon as the channel is busy. Since CSMA/CA is implemented at the slave nodes, this MAC component is the *slave_node_v3* component. The channel is considered busy when the *FlexframeSync* component is able to recognize a valid physical layer header. If the *FlexframeSync* component waits for the entire frame to be received before informing the subsequent MAC component, it induces an unwanted delay in carrier sensing process. Therefore, the *FlexframeSync* component must relay the information about a busy channel as soon as it realizes a valid header. In the existing design of the waveform the *slave_node_v3* component, farthest down the receive chain, will only be able to know the channel is busy after the *FlexframeSync* component has completely processed the whole frame with a valid header. As a result of this, a special interface has been defined between the *FlexframeSync* component and the *slave_node_v3* component to serve this purpose. The status of the channel is basically shared using special variables called ‘flags’ in a file which can be accessed by both the components.

#### 3.2.2.3 Squelch threshold

The *FlexframeSync* component provides an option ‘Squelch threshold’ which needs to be enabled while dealing with packet data. An incoming signal will be detected if and only its received signal strength is over the defined squelch threshold. The squelch threshold must be defined above the noise floor otherwise the receiver will not be able to synchronize with the header. The average noise floor value has been measured to be -33dB, which means the squelch threshold is defined at 3dB above the noise floor. It is set at -30dB so that it is about 3dbB above the measured noise floor.  If a squelch threshold option is not enabled, the *FlexframeSync* component will try to synchronize with any signal level, which includes below the squelch threshold. Since packet data arrives in bursts, during the gap between two consecutive packets, the receiver will try to synchronize with the noise in the absence of any detectable signal. When this happens, the phase locked loop is unable to lock on to any frequency and keeps deviating.

As a result of this, the next time a signal is detected it takes longer for the PLL to come to lock on to the frequency. This causes erroneous reception of a message thereby containing bit errors. This causes higher number of retransmissions thereby decreasing efficiency and throughput of the system. Once a valid PHY header is detected at *FlexframeSync* component it does not squelch anymore. What this means is that if the signal strength goes below -30dB in the duration of the payload of the frame, the PLL will still try to synchronize with the low signal level or noise. So the physical layer resets the PLL after a number of samples equivalent to the payload length and as specified in the PHY header.

#### 3.2.2.4 Role in rate adaptation

The *FlexframeGen* component plays an important role in rate adaptation. This component performs physical layer transformations on the data bits received from the preceding MAC component. The rate adaptation algorithm, implemented in the MAC component, determines a modulation scheme for the data to be sent out. The question that arises is how the subsequent *FlexframeGen* component can know which modulation scheme to adopt. This is made feasible by simply using a predefined array where each number is mapped to a particular type of modulation scheme. A MAC layer component simply passes this index number through the MAC header. The *FlexframeGen* component reads this index number and maps to the desired modulation type before initiating the physical layer transformations on the MAC datagram.

### 3.2.3 Support for packet data on OSSIE

OSSIE has a native support for streaming data (used in narrow band FM applications using FRS radios) but it is not practical for packet radio applications. In case of streaming data, a continuous block of data is sent while in the case of packet data, data is transmitted in bursts. The OSSIE USRP device (for USRP1) uses a device driver file, ‘usrp.cpp’, which is not suitable for packet data applications. In this context the terms ‘packet’ and ‘frame’ are interchangeable.

In order to understand the gravity of the underlying problem, let us first understand how the current OSSIE USRP device works. The USB 2.0 interface transmits data to or from the host in blocks of 512 bytes (at about 11 microseconds per block). The OSSIE device driver for USRP1 maintains a transmit buffer of size 512 bytes before pushing the samples over the USB. As soon as the USRP1 receives these USB blocks, it transmits them after the digital to analog conversion. In the current version of usrp.cpp, the OSSIE USRP device waits for this transmit buffer to be full before pushing the samples to the USB. This wait in filling the transmit buffer poses a significant problem to packet data transfer. Number of complex short samples in 512 bytes = 512/4 =128 (4 is the local interpolation rate used in this case). Therefore, the USRP device waits for 128 samples before pushing them to the USB. Figure 3.6 illustrates this problem with the help of an example.

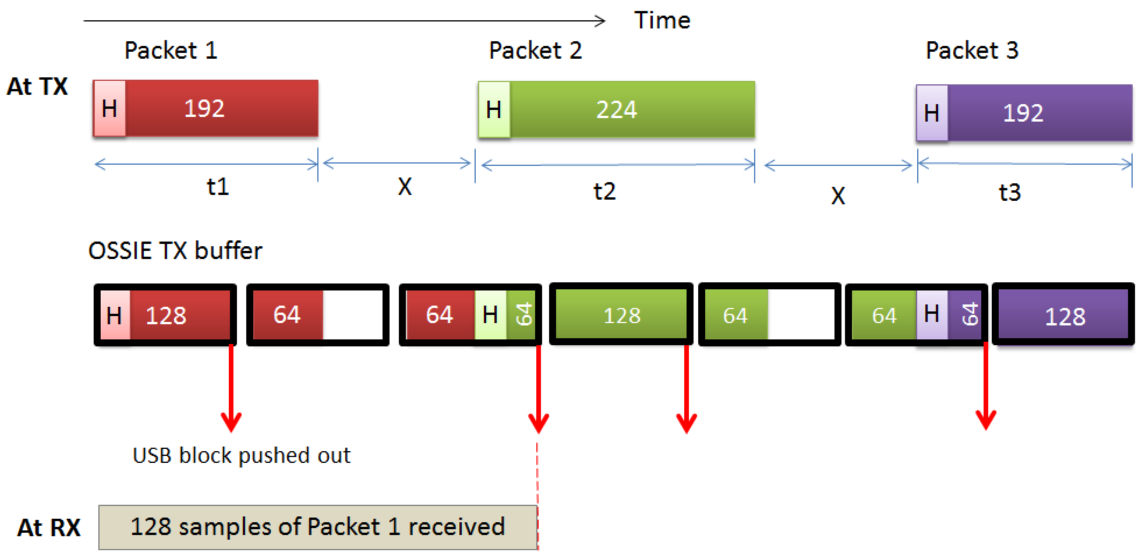


Figure 3.6: Packet data handling with original OSSIE USRP device driver

In Figure 3.6, H represents the header information of the packet which is used by the receiver to synchronize and identify the packet. The packets are sent out at an interval of X seconds. Contents of a packet are color coded for ease in understanding. Size of a packet is identified by the number on it, represented as number of complex short samples. As a packet is assembled and ready to be transmitted, chunks from the packets are put into USB blocks to be sent to the USRP. Since USB blocks carry 128 complex short samples (512 bytes) of information, the OSSIE

USRP device waits till it has collected 128 samples in the TX buffer before passing it to the USB. In this example, 128 samples are sent out to one USB block from the first packet while the remaining 64 samples are put in the TX buffer waiting to be full. It has to wait for at least X seconds before the transmit buffer is filled with a chunk from the second packet. By this time the first USB block has already been transmitted. It continues in this fashion till all the packets have been transmitted.

Assuming only the transmission delay, the receiver will receive first 128 samples of the first packet in around (t1/2) seconds. It however doesn't receive whole of packet 1. It also doesn't receive any of the subsequent packets because the blocks of data do not reach the receiver continuously because of transmit buffer wait delay. When these blocks of data reach the receiver at intervals, each such block behaves like an individual packet (or frame). The receiver can only receive them successfully if each such block has a recognizable packet header at the beginning. But clearly, from the figure above, the receiving side does not find any header at the beginning of each block of data. This will cause retransmissions but there is still no guarantee that the packets will be transmitted on their retransmission. So, we can conclude that if a packet size is not a multiple of the USB 2.0 block size this problem will occur and affect the throughput as well as efficiency.

This problem has been solved in this work by padding every residual chunk of data from a packet in a USB block to 128 samples so that the residue could be transmitted along with its rest of the packet. This ensures that the receiver will receive a whole packet in the form of continuous blocks of data. Figure 3.7 illustrates this solution with the help of the same example as used in Figure 3.6.

In the Figure 3.7, the USB blocks containing residues of packets 1, 2 and 3 are zero-padded and transmitted one after the other continuously. The receiving side receives these blocks one after the other continuously and as such, each whole packet is received. This clearly increases the efficiency and throughput of the system. The effect of this on the MAC protocol performance is shown in section 8.4.4 ahead.

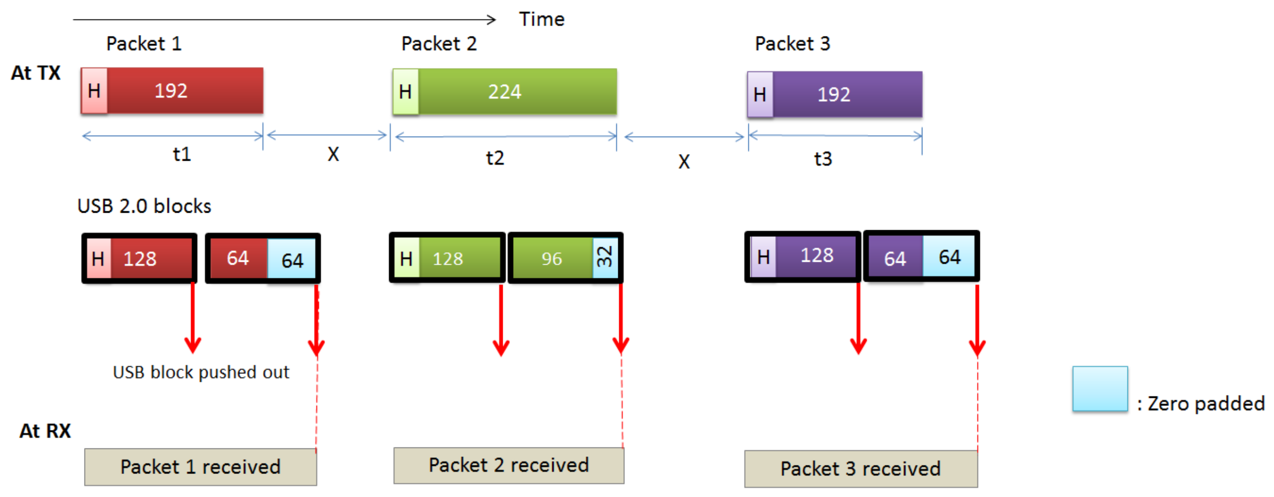


Figure 3.7: Packet handling with new OSSIE USRP device driver

# Chapter 4

# MAC Framework for WDC

## 4.1 Overview

A MAC protocol needs an underlying MAC framework, which defines a unanimous format for different types of messages exchanged between nodes. This chapter will explain the following main aspects of the MAC framework:

1) Message format and different types of messages.

2) Flags that distinguish a message from another.

3) Configuration of the protocol.

There are two types of messages in the WDC MAC protocol: data messages and control messages. A MAC layer frame has a header and a payload. A control message has been defined to consist of only the header while a data message consists of both the header and payload. The MAC header has been defined to be eight bytes long of the same size as an UDP header. The payload length can vary based on the modulation scheme, error detection scheme and FEC type used. The format of the frame is the same for uplink and downlink but the header elements may hold different interpretations owing to different multiple access schemes used on UL and DL. The DL protocol uses a round robin based polling channel access scheme while two alternative UL protocols have been designed using: (**I**) *CSMA/CA* for non-centralized data transfer and (**II**) *Polling*, respectively.

The different types of control messages used are:

1) Request for acknowledgement (from master to slave node) in DL data transfer

2) Acknowledgement (from slave to master node) in DL data transfer.

3) Request for UL data (from master to slave node) in UL data transfer using **II**.

4) Request to send (from slave to master node) in UL data transfer using **I**.

5) Clear to send (from master to slave node) in UL data transfer using **I**.

## 4.2 Message format

The format for the data message for both UL and DL is the same. It is necessary to design a format which allows this limited size header to be used resourcefully. Figure 4.1 shows the generic MAC datagram/frame structure.

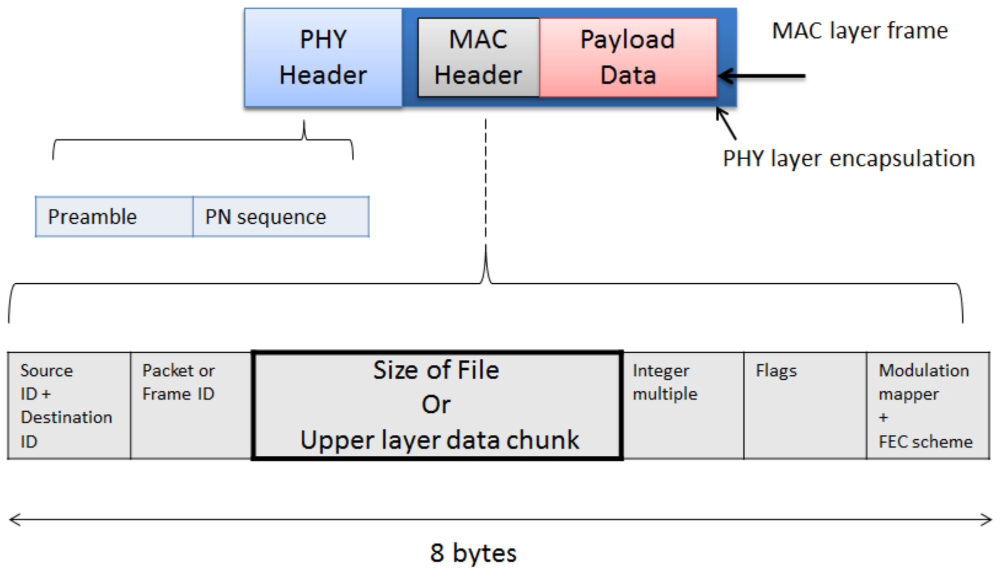


Figure 4.1: MAC frame format

The first byte (8 bits) represents the ***source or destination ID*** unique for each node. This byte is split in to two unequal parts: source node ID and destination node ID. The Master node is identified by node ID 0, where as a slave node can be identified by any other number lesser than 254. The node ID 254 is defined as a broadcast ID which can be received by all the slave nodes. At the master node, the source node ID is denoted by the rightmost bit while the destination node ID is denoted by leftmost 7 bits. At the slave node, the source node ID is represented by rightmost 7 bits while the destination ID is represented by the leftmost 1 bit. This means the slave node ID can vary from 1 to 127.

The figure below shows the representation of node ID in the MAC header.

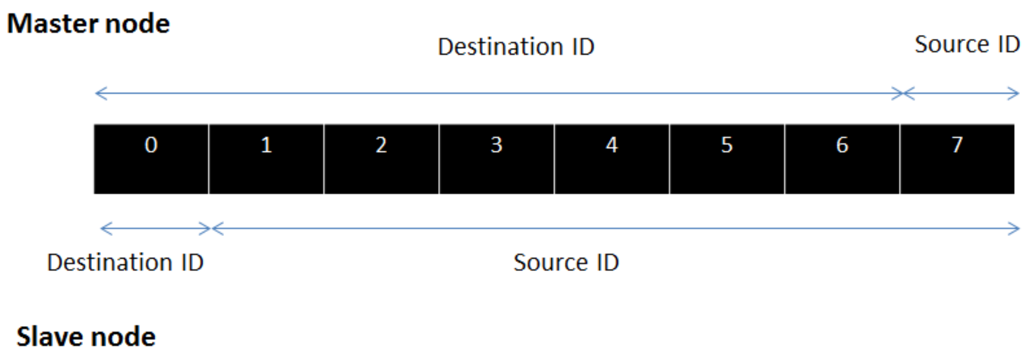


Figure 4.2: Numbering format of the nodes

The '***packet or frame ID***' element represents the sequence number of the packet or frame. It is used in reliable data transfer techniques (sliding window mechanism used in Go back N protocol) and helps maintain continuity of data. The maximum value of this element can be 255 which means, packet or frame ID is reused every 256 times.

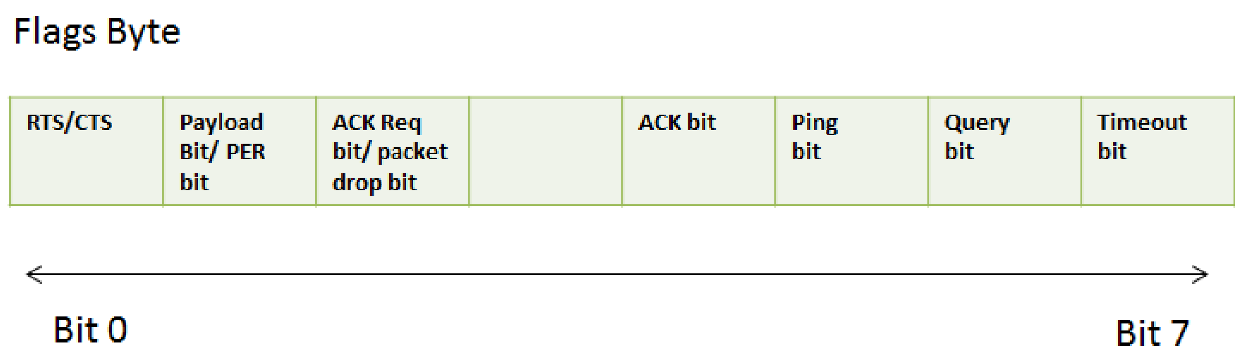
The '***size of file***' element represents the amount of data that can be transferred from a file or an upper layer data segment. This element occupies three bytes of data which means the maximum size of a file or an upper layer data chunk can be $2^{24}$ bytes = 16MB. If the file to be transferred is any larger than this size, it will be segmented into files of smaller sizes.

The '***integer multiple'*** element basically helps in determining the size of actual data stored in a frame. This finds use in the context of hybrid ARQ.

The ‘***modulation and FEC scheme mapper***’ element is used by the MAC layer component to convey to the subsequent *FlexframeGen* component in the transmit chain about the selected modulation scheme. The four most significant bits are used precisely for this purpose, which means 16 different modulation schemes can be chosen. The remaining four bits have been assigned to carry information regarding the FEC type. This means sixteen possible error correction coding schemes can be represented. The contents of this element are determined by the rate adaptation algorithm discussed later.

The ‘***flags***’ element carry flags and indicator variables which are useful in identifying the different types of messages in a protocol. It is 8-bit long.

The structure of ‘flags’ element is shown below:


Flags Byte
Bit 0
Bit 7

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| RTS/CTS | Payload Bit/ PER bit | ACK Req bit/ packet drop bit | | ACK bit | Ping bit | Query bit | Timeout bit |

Figure 4.3: Flags in the MAC header

From the above figure, bit numbers 1, 2, 4, 6 and 7 have been used for DL data transfer, whereas bit numbers 0, 1, 2, 5, 6 and 7 have been used for UL data transfer (using both **I** & **II**). The *RTS/CTS* bit marks whether the message is an RTS (RTS= request to send; from slave to master) or CTS (CTS= Clear to send; from master to slave). If set, it means CTS otherwise RTS. This is used in UL protocol (**I**). The *Payload bit*, when set, signifies that the header is followed by a payload; it is always set in a data message in UL or DL protocol. This bit is alternatively used in an acknowledgement message in DL protocol and UL protocol (**II**) to indicate a frame has been discarded on the destination node due to packet errors. The *ACK Request bit*, if set, denotes that the master node is requesting a slave node to send an acknowledgement; it is used in DL protocol. This bit number is alternatively used in an acknowledgement message to indicate a frame drop at the destination node. The *ACK bit*, if set, characterizes an acknowledgement from the slave node during DL data transfer. The *Ping bit*, if set, indicates a request for UL data from the master node to a slave node in the UL protocol (**II**). The *Query bit*, used in both UL and DL protocols, represents the direction of transfer of a message. If this bit is set, it means that the message is sent from a slave to master node otherwise vice versa. The *Timeout bit*, if set, is used only in an acknowledgement message in both UL and DL protocol to identify that a timeout had occurred at the destination node while receiving the last frame (within a window). Any unused bit number can be used in the future.

When a node transmits a message and immediately switches to the receive mode, due to leakage in the USRP hardware, it can receive its own message. The query bit and the source node ID are used to identify a leaked signal. In case, a leaked signal is detected, it is simply discarded. If a master node transmits a message, only the query bit in the header is required to realize whether it is a leaked transmitted signal or not. But in case of a slave node, just checking the query bit will not suffice as the signal could very well originate from a different slave node. As a result of this, the source node ID is also checked to verify if the received signal is a leaked transmitted signal or not.

## 4.3 Types of Messages

The Tables 4.1-4.7 show the combinations of header elements and their values that enable a receiving node to distinguish between different types of messages in the MAC protocol. The right column represents the header element and the left side represents the status of the element for a particular type of message.

Table 4.1: Request for Acknowledgement from master node to slave node

| | |
|---|---|
| Destination ID | Slave node ID(1 to 127) |
| Source ID | 0 |
| Query bit | Not set |
| Payload bit | Not set |
| ACK Req bit | Set |

Table 4.2: DL data from master node to slave node

| | |
|---|---|
| Destination ID | Slave node ID(1 to 127) |
| Source ID | 0 |
| Query bit | Not set |
| Payload bit | Set |
| ACK Req bit | Not Set |

Table 4.3: Request for UL data (UL multiple access scheme (**II**))

| | |
|---|---|
| Destination ID | Slave node ID(1 to 127) |
| Source ID | 0 |
| Query bit | Not set |
| Ping bit | Set |
| PER bit | depends on channel conditions |
| Packet drop bit | depends on channel conditions |
| Timeout bit | depends on channel conditions |

Table 4.4: UL data (UL multiple access scheme (**I &II**))

| | |
|---|---|
| Source ID | Slave node ID (1 to 127) |
| Destination ID | 0 |
| Payload bit | Set |
| Query bit | Set |

Table 4.5: RTS (UL multiple access scheme (**I**))

| | |
|---|---|
| RTS/CTS bit | Not set |
| Source ID | Slave node ID (1 to 127) |
| Destination ID | 0 |
| Payload bit | Not set |
| Query bit | Set |

Table 4.6: CTS (UL multiple access scheme (**I**))

| | |
|---|---|
| RTS/CTS bit | Set |
| Source ID | 0 |
| Destination ID | Slave node ID (1 to 127) |
| PER bit | depends on channel conditions |
| Packet drop bit | depends on channel conditions |
| Timeout bit | depends on channel conditions |
| Query bit | Not set |

Table 4.7: Acknowledgement for DL data

| | |
|---|---|
| Destination ID | 0 |
| Source ID | Slave Node ID (1to 127) |
| Query bit | Set |
| ACK bit | Set |
| PER bit | depends on channel conditions |
| Packet drop bit | depends on channel conditions |
| Timeout bit | depends on channel conditions |

## 4.4 Code structure

As explained in the previous chapter, the code within the MAC layer components is divided into two threads: RX thread and TX thread. The RX thread has access to the input ports while the TX thread has access to the output ports of the component. Communication between the two threads is established using global variables in the common user space. Each thread contains the code that implements several algorithms in UL and DL protocols. The code is organized into smaller sections of code called modules, each module identified by a combination of flags and indicator variables. Each module is further divided into sub-modules. These modules and sub-modules essentially represent different modes of operation with specific functionalities. We will now study the code structure.

The highest level modules in a MAC layer component are:

- *TX mode*: In this mode, a node is configured to transmit a MAC datagram.
- *RX mode*: In this mode, a node is configured to receive a MAC datagram.
- *Reset mode*: In this mode a node resets itself by initializing all variables, parameters and flags used.

*TX mode* is implemented within the TX thread while *RX mode* and reset mode are implemented within the RX thread.

Within the *TX mode,* there are several sub modules, which are as follows:

<u>At the master node</u>,

- *Ping for ACK mode*: The master node requests the slave node for an acknowledgement on DL data transferred.
- *Send data mode*: The master node sends DL data to the slave node.
- *Ping for UL data mode*: The master node requests the slave node for UL data (DL multiple access scheme **II**).
- *Send CTS mode*: The master node sends a CTS in reply to an RTS (DL multiple access scheme **I**).

At a slave node,

- *Send ACK mode*: The slave node sends an acknowledgement to the master node.
- *Send data mode*: The slave node sends UL data to the master node (UL multiple access schemes **I** & **II**).
- *Send RTS mode*: The slave node sends RTS to the master node. (UL multiple access scheme **I**).

Within the *RX mode*, there are several sub modules as well, which are as follows-

At the master node,

- *Receive ACK mode*: The master node receives acknowledgement from the slave node during DL data transfer.
- *Receive data mode*: The master node receives UL data from slave node (UL multiple access schemes **I** & **II**).
- *Receive RTS mode*: The master node receives RTS from a slave node (UL multiple access scheme **I**).

At the slave node,

- *Receive ACK request mode*: The slave node receives the master node's request for an acknowledgement during DL data transfer.
- *Receive UL data request mode*: The slave node receives the master node's request for UL data during UL data transfer (UL multiple access scheme **II**).
- *Receive data mode*: The slave node receives DL data from the master node.
- *Receive CTS mode*: The slave node receives CTS from the master node during UL data transfer (UL multiple access scheme **I**).

## 4.5 Sequence of events

Figure 4.4 summarizes the overall working of the MAC protocol by demonstrating the sequence in which the modules and sub-modules get activated. In this figure, the numbers denote the sequence of events occurring in DL and UL, while the color represents the UL and DL protocols. Each rectangular block represents a sub-module explained above. TX mode and RX mode represent the processing threads the sub-modules belong to. These sequences of events constitute the overall MAC protocol. The events shown here do not necessarily occur one after the other. For example, the slave node keeps receiving data simultaneously as the master node keeps sending data frames. The sequence number simply represents an event triggering a subsequent event.

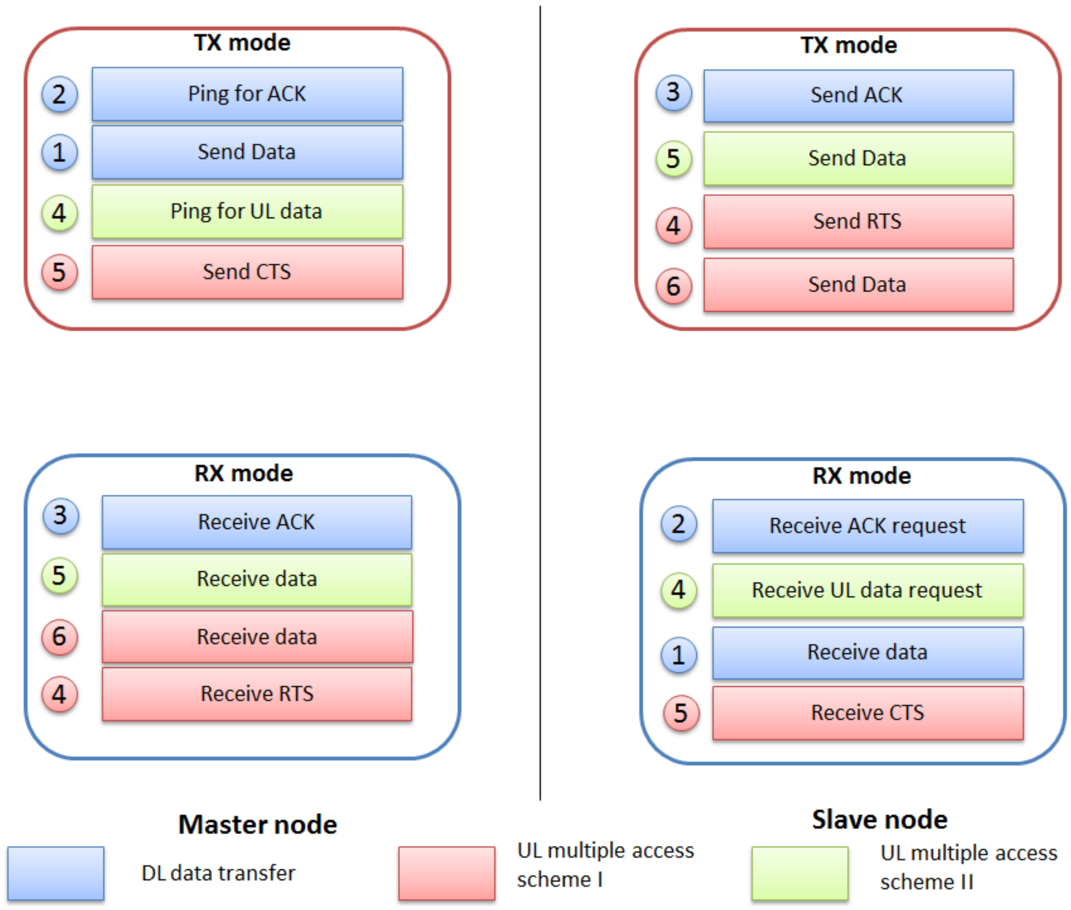


Figure 4.4: Sequence of events in the Uplink protocols

# Chapter 5

# Downlink Protocol

## 5.1 Rationale

Downlink (DL) refers to the transfer of messages (data or control) from the master node to the slave node. According to the paradigm of wireless distributed computing, the master node has to communicate with multiple slave nodes on the downlink. It is quintessential to be able to implement wireless distributed computing on a wide spectrum of radios with different capabilities, which includes radios with limited hardware resources that are cost effective. For this reason, we considered low budget half-duplex USRP1 hardware as our radio front end devices. Since the radio devices are half-duplex, the master node can only use the common antenna and the channel for either transmitting or receiving to or from one slave node at a time. It is transparent that the DL protocol will need to employ a channel access scheme which enables the master node to communicate with each of the slave nodes. Consequently, a channel access scheme which schedules transmissions in a round robin fashion befits this requirement. During the DL transmission of data, the master node should also be able to get feedback from a slave node to ensure reliability. As explained in section 2.3, in the context of WDC, it is desirable to be able to include reliable data transfer techniques, rate adaptation mechanism and link failure recovery mechanism in the DL MAC protocol. It is now apparent that the DL MAC protocol should be able to meet all these expectations.

## 5.2 Polling protocol

Polling protocol uses a polling channel access scheme. Polling basically refers to a node communicating with other nodes in a round robin manner. This is very suitable for a star topology based WDC network. In the context of WDC, the master node has to send different set of data to the different slave nodes on the downlink. The information regarding the number of slave nodes in the WDC network is assumed to be predefined at the master node in this work. A link refers to the communication between the master and a particular slave node for exchange of messages. In this channel access scheme, a slave node is polled at an interval of small fragment of data called frame. The slave nodes are polled based on a sequence of their node ID till data transfer on all the links is complete. A link involving a slave node is independent of the other. This implies that one slave node can finish receiving data on a link before or after the other slave nodes. Once the data transfer on a particular link completes the master node doesn't poll that slave node any further while preserving the order of the round robin polling.

This DL protocol also schedules acknowledgements in the reverse direction and incorporates reliable data transfer mechanism on each link. It is quite intuitive that if the master node waits for an acknowledgement from a slave node after every data frame, it will take a long time to complete data transfer compared to waiting for acknowledgement every few number of data frames. This is also proven in section 8.4.2. Consequently, the master node sends a 'window' of frames to a slave node before requesting for an acknowledgement. A window refers to a group of frames transmitted in a sequence but not necessarily continuously. The size of this window can be different for every slave node and is controlled by the rate adaptation algorithm to improve the performance of the protocol under varying channel conditions. An acknowledgement (ACK) from a slave node is always preceded by a request for acknowledgement (RACK) from the master. A RACK is used mainly due to the reason that the slave node is oblivious of the window size chosen by the master node.

While a 'window' controls the number of frames that can be transmitted before requesting for an acknowledgement, the sliding window mechanism [11] determines the sequence numbers of frames to be sent in a window. Another mechanism verifies the data in those frames for continuity while the rate adaptation algorithm determines the rate.

The following figure demonstrates the overall working of the downlink protocol with the help of an example.

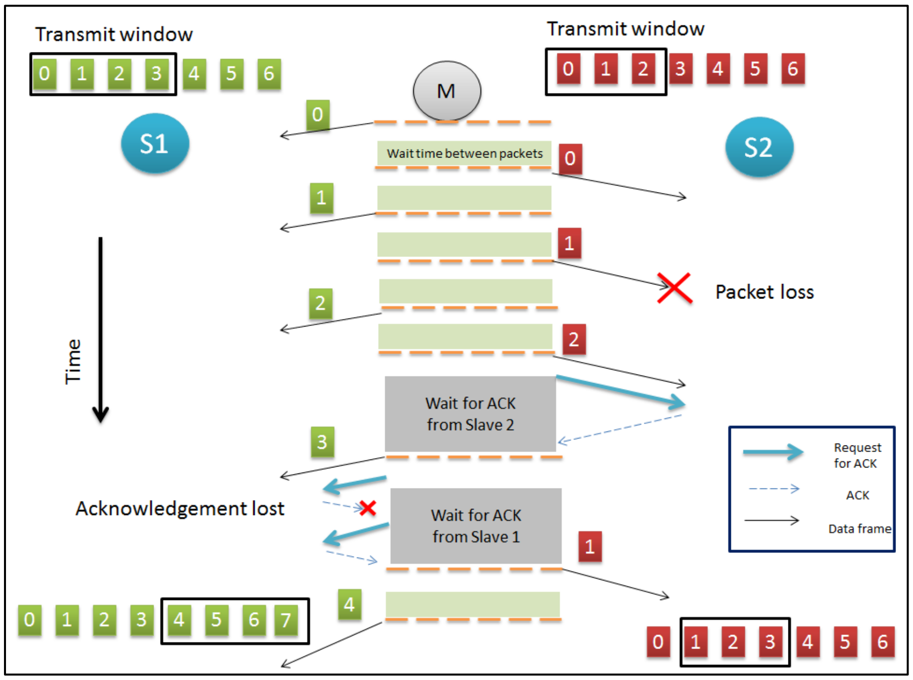


Figure 5.1: Downlink polling protocol working

The above figure shows a WDC network with a master node and two slave nodes. The master node is sending DL data to the slave nodes. The master node selects a window size of four for *S1* and a window size of three for *S2*. The master node sends a frame from one window to *S1* and then sends a frame from the other window to *S2*. At the end of each window, the master node requests the corresponding slave node for an acknowledgement. The master node accepts cumulative acknowledgements as per the sliding window protocol.  In this example, the acknowledgment from *S1* is not received by the master node within a defined wait time. So, the master node sends another request for acknowledgement to *S1* and then receives the acknowledgement. The master node sends the next window to *S1* starting with the frame bearing sequence number 4. In this example *S2* couldn't receive the frame with sequence number 1 in its first window. As a result, *S2* acknowledged the last successfully received frame bearing sequence number 0. Consequently, the master node sends the next window to *S2* starting with the frame bearing sequence number 1. This is how the sliding window mechanism has been incorporated.

The following figure represents the time domain scheduling during DL data transfer from the above example.

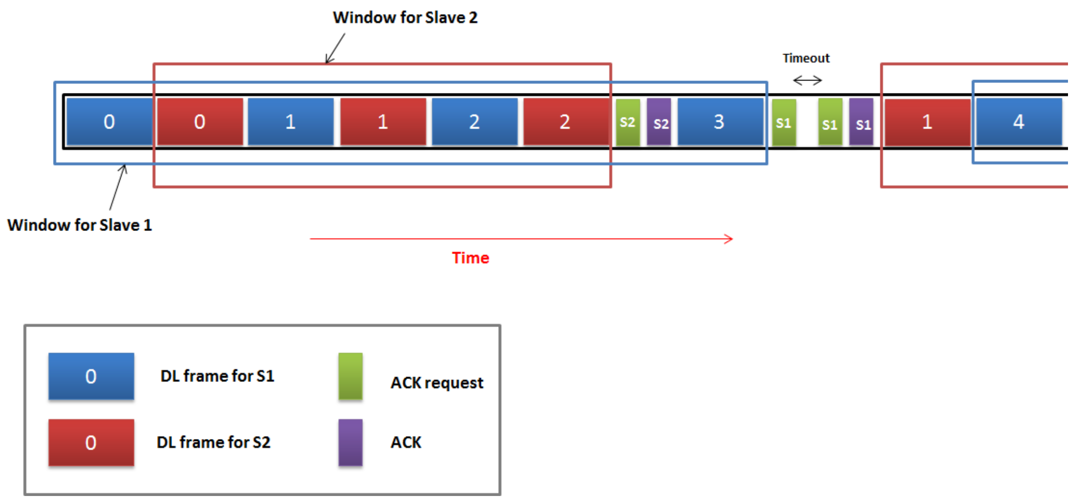


Figure 5.2: Time domain scheduling in downlink

In the above figure, it is to be noted that the windows for different slave nodes overlap in time but the individual frames do not. The practical time domain representation of downlink data transfer as obtained from a spectrum analyzer is annotated and shown in the Appendix.

## 5.3 Design and implementation

Table 5.1 represents the naming conventions or symbolic names used in explaining the implementation and working of the DL protocol at the master node. The left column lists the names while the right column describes them.

Table 5.1: Naming conventions used in the Downlink protocol at the master node

| | |
|---|---|
| next_fid [k] | Sequence number of the first frame of the next window to be sent to the kth slave node. |
| curr_fid[k] | Sequence number of the frame to be sent to kth slave node. |
| last_fid [k] | Sequence number of the most recently sent frame to kth slave node. |
| rcv_fid[k] | Sequence number of the frame requested by the kth slave node. |
| send_win[k] | Current window size for the kth slave node. A window consists of multiple frames. |
| last_send_win[k] | The size of the most recent window sent to the kth slave node. |
| same_frame_count[k] | The number of times the same window of frames has been sent successively (identified by sequence number of the first frame in a window). |
| same_frame_count_limit[k] | The maximum number of times the same window of frames can be sent to the kth slave node before data transfer is discontinued. |
| win_count[k] | The number of frames sent to the kth slave node in the current window. |
| flag[[k] | A flag, when set, denotes completion of data transfer to the kth slave node. |
| file_end[k] | A flag, when set, denotes no more data left to be sent to the kth slave node. |
| l_win_beg [k] | Sequence number of the first frame in the previous window. |
| byte_counter[k] | The number of bytes successfully received by the kth slave node. |
| pay_len[k][m] | The size of mth frame in the previous window sent to the kth slave node. |
| f_id[k][m] | The sequence number of the mth frame in the previous window sent to the kth slave node. |

## 5.3.1 Handling of data

This section talks about the operations that take place at the master node during DL data transfer. As stated previously, the master node has knowledge of the number of slave nodes ( $N_{sl}$ ) in the network. The different modes of operation of the master node are described in section 3.3. The master node switches between TX mode and RX mode in course of data transfer.

We now know that the master node polls a slave node as per a round robin sequence at the end of every data frame. This operation is carried out in a loop till the overall DL data transfer is complete. At the beginning of every iteration of this loop, the master node checks the values of *flag[k]*, *win_count[k]* and *file_end[k]* in the TX mode. Before preparing a frame of data, the master node first checks whether DL data transfer is complete on all the links. If so, there is no further DL data transfer, otherwise the master node verifies using *file_end[k]* whether there is any data left to be transferred on the kth link. In case there is no more data left to be transferred on that link, the master node simply polls the next slave node in sequence.

Therefore, conditions for *Send data mode* at the master node:

- flag[k] is not set.
- win_count[k] < send_win[k], or, file_end[k] is not set.

Once these conditions are met, the master node determines the sequence number of the next frame to be transmitted to its corresponding slave node using the following expression:

$$\text{curr\_fid}[k] = (\text{last\_fid}[k] + 1) \text{ AND } 0\text{xff}_{\text{HEX}}.$$

AND represents logical AND. This means that every 256 times *curr_fid[k]* will be reset to 0. This helps in reusing frame ID numbers after 256 times.

In the *Send data mode*, the master node reads data from a specific location in a file (identified by a file pointer) and assembles a frame before sending it out. The position of the pointer keeps sliding as each byte of data is put into the frame. As the current frame destined for the kth slave node is pushed out of the MAC component, its sequence number '*curr_fid[k]*' is stored in the *f_id[k][m]* while its data size is stored in *pay_len[k][m]*. *last_fid[k]* is updated with the value of *curr_fid[k]*. After the frame is transmitted, *win_count[k]* is incremented by one to record the number of frames sent from a particular window. These actions are performed every time a master node has to send a data frame. Every time the first frame of a window (win_count[k] = 0) is pushed out, the sequence number of this frame is recorded in *l_win_beg[k]*. This is done because just prior to the next time the master node sends a new window of frames to the same slave node, the sequence number of the first frame in the window is compared to the current value of *l_win_beg[k]*. If they match, it implies that the previous window of frames could not be received by the slave node.

When all the frames in a window are transmitted (i.e. win_count[k] = send_win[k]), the master node requests for an acknowledgement in *Ping for ACK mode*. This is also possible if there is no data left to be transferred (i.e. *file_end[k]* is set) on that link. The size of the most recently sent window (for $k^{th}$ slave node) is stored in *last_send_win[k]* and the window count is reset.

After the master node has sent a RACK to a particular slave node, it waits for an acknowledgement from the corresponding node in the *Receive ACK mode* for $T_{ping}$ seconds, which must be greater than the round trip time for a control message. If the master node does not receive an acknowledgement in this period (i.e. timeout) it can send another RACK. The master node can send a RACK for a defined maximum number of times (denoted by $N_{ping}$ ) before it polls the next slave node in sequence. This means that the master node has to wait a total duration of ($N_{ping}$ . $T_{ping}$) before polling the next slave node. During the *Receive ACK mode*, the master node receives acknowledgements only from the right slave node. On receiving an acknowledgement from the $k^{th}$ slave node, the master node updates *next_fid[k]* with the sequence number of the next frame requested by the slave node *rcv_fid[k]*. The acknowledgement also carries information about whether any frame was discarded at the slave node due to errors or whether any frame was dropped at the slave node. This information is used in the rate adaptation algorithm.

Figure 5.3 represents a high level flowchart showing the aforementioned operations that take place at the master node during the downlink data transfer. The downlink MAC protocol uses this algorithm bind together message scheduling, reliable data transfer and rate adaptation. In Figure 5.3, ACK_req is a flag, when ON, represents *Ping for ACK mode.* Similarly, ACK_mode is a flag, when ON, represents *Receive ACK mode.* The symbol '||' denotes a logical OR. The green boxes highlight the actions taken while the grey boxes denote conditions checked.

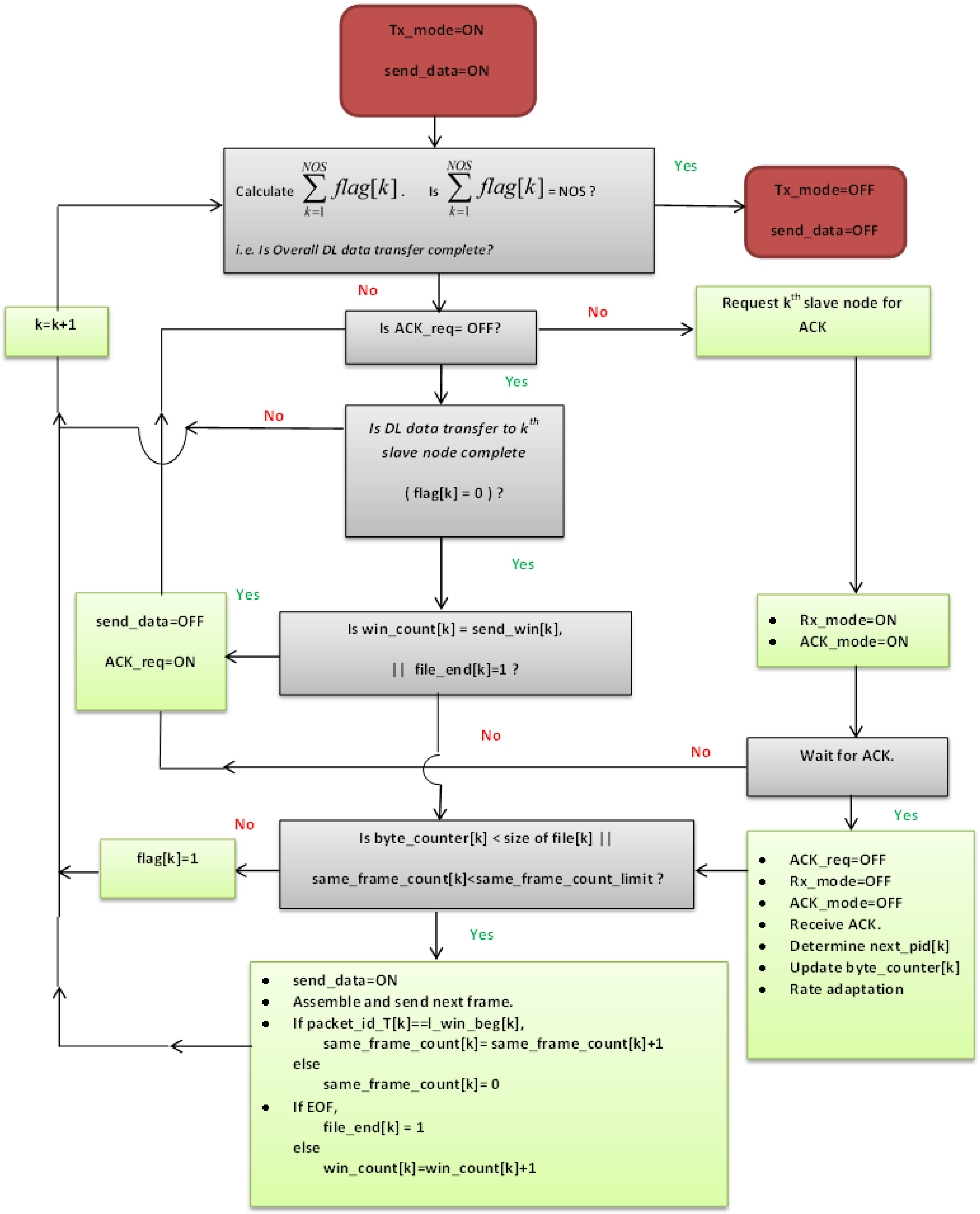

Tx_mode=ON
send_data=ON
Calculate $\sum_{k=1}^{NOS} flag[k]$. Is $\sum_{k=1}^{NOS} flag[k]$ = NOS ?
i.e. Is Overall DL data transfer complete?
Yes
Tx_mode=OFF
send_data=OFF
No
k=k+1
Is ACK_req= OFF?
No
Request k$^{th}$ slave node for ACK
Yes
No
Is DL data transfer to k$^{th}$ slave node complete ( flag[k] = 0 ) ?
Yes
Yes
send_data=OFF
ACK_req=ON
Is win_count[k] = send_win[k], || file_end[k]=1 ?
Rx_mode=ON
ACK_mode=ON
No
No
Wait for ACK.
No
flag[k]=1
Is byte_counter[k] < size of file[k] || same_frame_count[k]<same_frame_count_limit ?
Yes
ACK_req=OFF
Rx_mode=OFF
ACK_mode=OFF
Receive ACK.
Determine next_pid[k]
Update byte_counter[k]
Rate adaptation
Yes
send_data=ON
Assemble and send next frame.
If packet_id_T[k]==l_win_beg[k], same_frame_count[k]= same_frame_count[k]+1 else same_frame_count[k]= 0
If EOF, file_end[k] = 1 else win_count[k]=win_count[k]+1

Figure 5.3: Data handling at master node during downlink data transfer

## 5.3.2 Reliable data transfer

While the sliding window mechanism makes sure that the sequence of frames is right, the data continuity mechanism makes sure the order of data is right across the frames. As we will see, this is especially required in retransmissions. Both these mechanisms together ensure reliable data transfer. The DL protocol exercises this reliable data transfer mechanism prior to starting a new window of frames for a particular slave node.

The reliable data transfer mechanism is explained in the following steps:

i. The master node checks *next_fid[k]* to send or resend a frame. As mentioned in the previous section, the value of *next_fid[k]* is determined by a cumulative acknowledgement received from the k$^{th}$ slave node. If a whole window of frames is received successfully by a slave node, then after the next update of *next_fid[k]*,

$$\text{next\_fid[k]} = (\text{l\_win\_beg[k]} + \text{last\_send\_win[k]}) \text{ AND } 0\text{xff}_{\text{HEX}}.$$

   This means that the master node does not need to retransmit any frame from the previous window.

   If, however, the slave node is not able to fully receive a window of frames from the master node, after the next update of *next_fid[k]*,

$$\text{next\_fid[k]} \neq (\text{l\_win\_beg[k]} + \text{last\_send\_win[k]}) \text{ AND } 0\text{xff}_{\text{HEX}}.$$

ii. The master node knows the sequence number of the frame requested but it still has to reassemble the data from the right location in the file. In order to do so, it incorporates the data continuity mechanism. In this mechanism, the master node looks for the sequence number that matches *rcv_pid[k]* in the *f_id* array of the corresponding slave node. Every time the master node comes across a sequence number from the previous window, the file pointer is simultaneously moved back by the size of the corresponding frame in the *pay_len* array located at the same index as *fid* array. This positions the file pointer to a position in the file from which the data for the next requested frame can be read.

iii. Based on the acknowledgement from the slave node, the master node updates the number of successful bytes transferred to the respective slave node. When the total number of data bytes successfully transferred on a link equals the size of the respective file being transferred, data transfer on that link is complete.

So, the f_id and pay_len arrays play an important role in reliable data transfer mechanism. The following two scenarios (a) and (b) explain how these arrays are filled during a window transfer.

a. Scenario1: When a full window is transmitted.

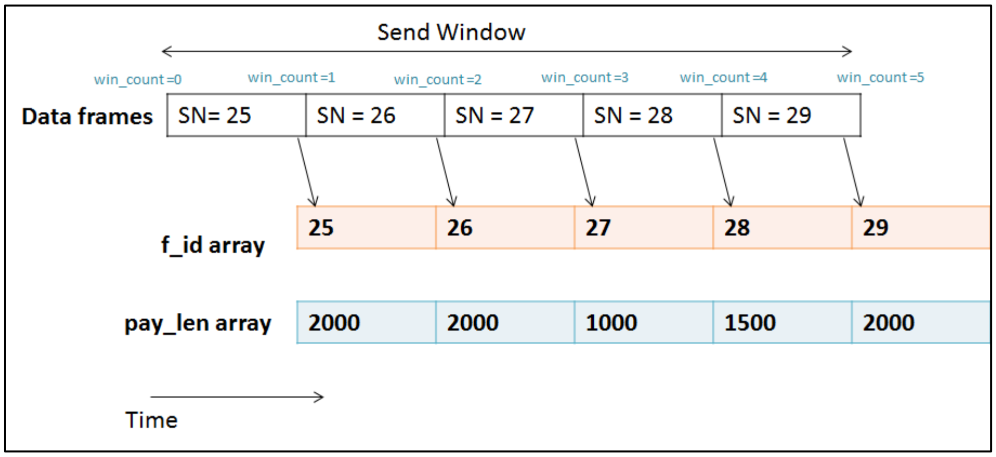


Figure 5.4: Storing frame information in Scenario 1

The above figure considers an example where frames with sequence numbers (SN) 25, 26, 27, 28 and 29 were sent in a window of size five. As each frame is transmitted, the *f_id* and *pay_len* arrays for the kth slave node are filled. win_count[k] is incremented after a frame is sent. After the complete window is sent (i.e. *win_count[k]* = *send_win[k]* = 5), *last_send_win[k]* is updated with the size of most recently sent window i.e. five.

b. Scenario 2: When a partial window is transmitted.

Figure 5.5 shows a scenario where there is no more data left to be sent in the middle of a window.

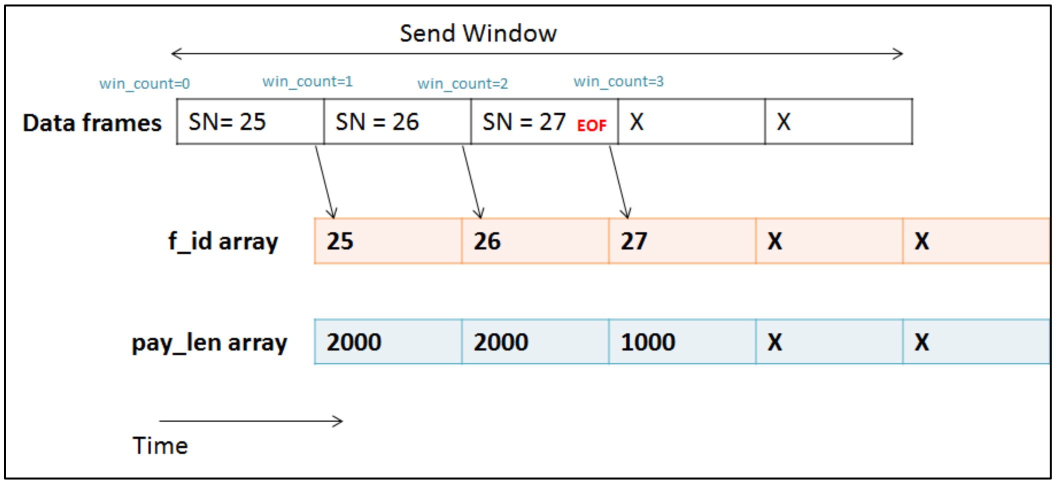


Figure 5.5: Storing frame information in Scenario 2

As shown in the figure above, the value of *last_send_win[k]* is updated with 3 after the final frame is sent.

The elements of *f_id* and *pay_len* arrays are accessed in case of retransmission. The following figures illustrate this during each of the above listed scenarios (a) and (b).

a. <u>Scenario 1</u>: When a complete window is transmitted.

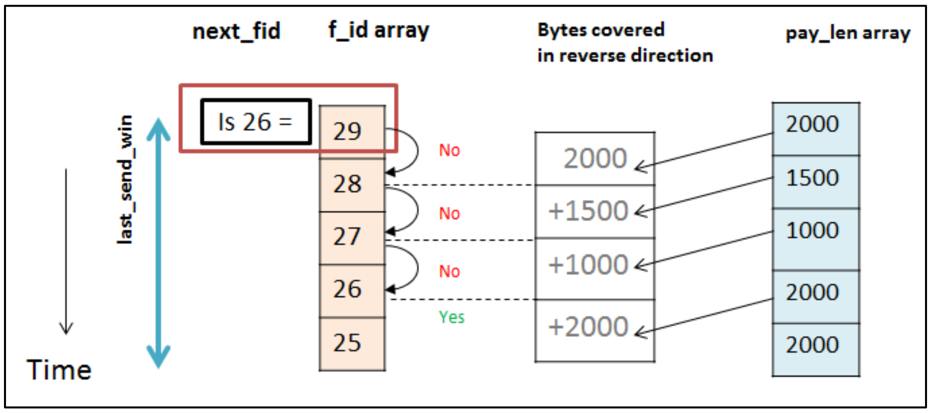


Figure 5.6: Retrieving Frame information in Scenario 1

In the above figure, *next_fid[k]* is matched against each element of f_id array. Before even starting the search, the master node checks *last_send_win[k]* to determine the number of frames that were actually transmitted to the kth slave node in the previous window (5 in this example).

As each element of *f_id* array is accessed, the file pointer is moved in the reverse direction by the size of the respective frame. In this case, in order to retransmit the frame with sequence number 26, the file pointer is moved backward by 6500 characters.

b. <u>Scenario 2</u>: When a partial window is transmitted.

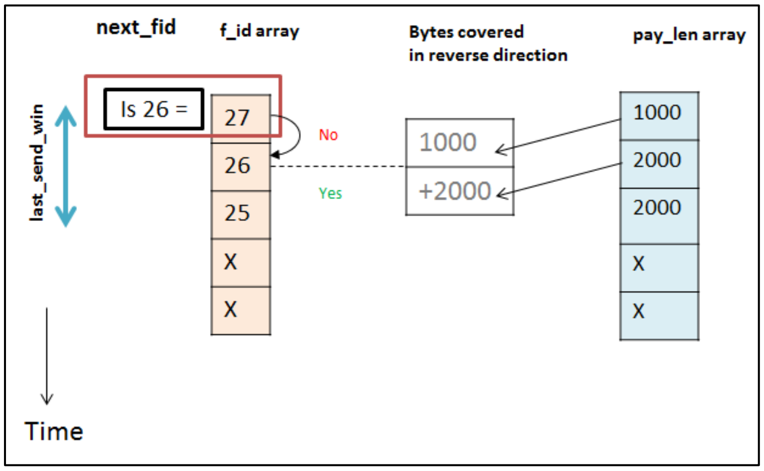


Figure 5.7: Retrieving Frame information in Scenario 2

In this case, *last_send_win[k]* is 3. So, in order to retransmit the frame with sequence number 26, the file pointer is moved backward by 3000 characters.

### 5.3.3 Role of slave node in downlink data transfer

The downlink protocol is realizable only when the master node and the slave node work together. We have already seen the master node side of this process. Now, let us take a look at what goes on at the slave node during downlink data transfer. Table 5.2 represents the naming conventions or symbolic variables used to explain the working of this DL protocol at the slave node. The left column lists the names while the right column describes them.

Table 5.2: Naming conventions used in the Downlink protocol at the slave node

| | |
|---|---|
| prev_fid | Sequence number of the last frame received successfully from the master node. |
| next_fid | Sequence number of the next expected frame from the master node. |
| frame_id_R | Sequence number of the current frame received. |
| nob | Number of bytes received successfully from the master node |

### 5.3.3.1 Data handling

A slave node is in the receive mode by default. *next_fid* is initialized to zero because the sequence number of the first frame that the slave node will receive from the master node starts from zero. A DL data frame sent from the master node is only accepted when *frame_id_R* = *next_fid*. Once a frame is accepted, it is decoded and PER is calculated. If the PER is within the acceptable QOS threshold $PER_{qos}$, the data is passed on to an output file or the upper layer. The following conditions affect the aforementioned variables as follows:

a) $PER < PER_{qos}$
next_fid = (next_fid+1) AND $0xff_{HEX}$.
nob = nob + 'message size in a frame'.
prev_fid = frame_id_R

b) $PER > PER_{qos}$
*i) nob > 0*
next_fid = (prev_fid+1) AND $0xff_{HEX}$
*ii) nob = 0*
next_fid = 0

The slave node has a 'receive' window of size one. Case (a) denotes implies that this receive window slides forward only when a frame is received with acceptable PER. The number of bytes successful received is also updated. If however, a frame has a PER above the defined threshold, it is discarded and the slave node expects a retransmission. Case (b) states that as long as the very first data frame does not pass the PER threshold, *next_fid* remains unchanged at zero. Otherwise, it requests the next frame.

The following figure demonstrates with an example how the variables *next_fid* and *prev_fid* are changed.

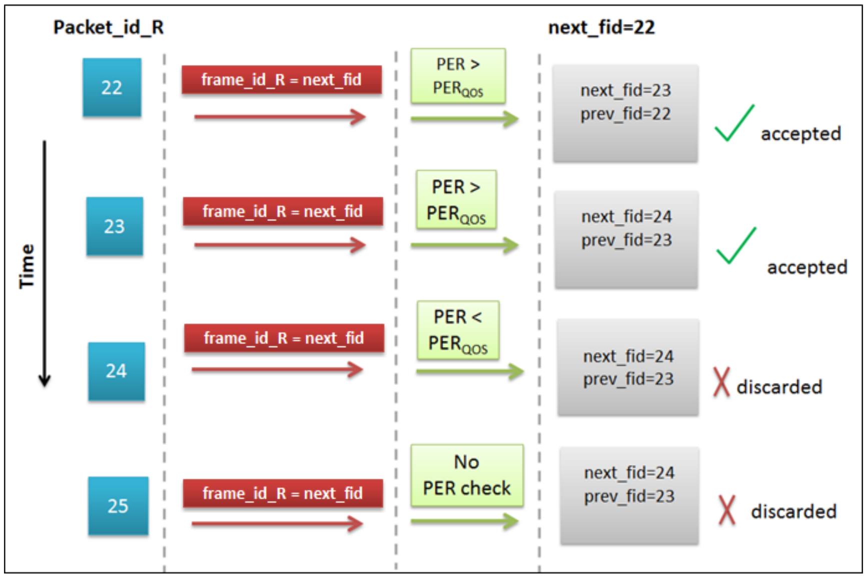


Figure 5.8: Data handling at slave node in the downlink

In the above figure, send window size at the master node is four. *next_fid* at the slave node starts with the sequence number 22 at the beginning of the send window. The frame is decoded only if *next_fid* = *frame_id_R*. If the frame is decoded correctly, *next_fid* and *prev_fid* are updated. It is to be noted that the slave node has no knowledge of the window size used at the master node. To make the job of a slave node easier, the downlink protocol mandates the slave node to send an acknowledgement message to the master node in reply to a request for acknowledgement.

### 5.3.3.2 Discrepancy in sequence numbers

The slave node plays an important role in providing feedback about DL data transfer to the master node through the acknowledgement. When *next_ fid* does not match *frame_id_R*, a discrepancy is detected at the slave node. When the master node receives an acknowledgement and realizes that the previous window was not received successfully at the slave node, it does not know the reason. This could be either of erroneous frames or frame drops at the slave node. In order to help the master node identify between the two, special flags are used to carry this information in the acknowledgement.

If ($PER > PER_{QOS}$), the flag *per_bit* is set. Whenever *next_fid* doesn't match *frame_id_R*, first the *per_bit* flag is checked to see if an erroneous frame has already been detected which could be causing this discrepancy. If the *per_bit* flag is not yet set but a discrepancy in sequence numbers is detected, it implies that a frame has been dropped. This is when the *p_drop* flag is triggered at the slave node. Once these flags are set, there should be a way to reset them otherwise every ACK frame will falsely denote a frame dropped or discarded. So, these flags are reset the next time a frame with expected sequence number is received at the slave node (i.e. *frame_id_R* = *next_fid*). When the master node requests the slave node for an acknowledgement, the slave node puts *next_fid*, *p_drop* and *per_bit* in the ACK frame.

## 5.4 Downlink Failure

This section highlights the actions taken at the master node as well as slave node to cope with link failure during downlink data transfer.

### 5.4.1 Master node

We know that at the end of every window, the master node switches to *Ping for ACK mode*, during which it requests a particular slave node for acknowledgement. After sending the request, it switches to *Receive ACK mode* to receive acknowledgement from that slave node. If the master node doesn't receive an acknowledgement from a particular slave node within the defined time duration $N_{ping}.T_{ping}$ , it retransmits the frames with the same sequence number in the next window when the same slave node is polled. Retransmission of a window can also occur because none of the frames could be received by the slave node either due to frame errors or first frame dropped. Every time the same window is retransmitted, the counter *same_frame_count[k]* is incremented by one for the k$^{th}$ slave node. When this counter reaches a defined value *same_frame_count_limit*, the master node ceases to send any further data on that link. This prevents the master node from sending the same window over and over again in case of link failure.

### 5.4.2 Slave node

If a slave node does not receive data from the master node due to link failure, the slave node will keep waiting indefinitely. In order to prevent this, the slave node starts a timer after receiving any message (control or data) from the master node. The slave node waits for a defined maximum wait time before terminating the data transfer process. In this case, the slave node makes the best out of whatever data has been received successfully. If the received data is insufficient for the slave node to process, the slave node can convey this to the master node during uplink data transfer. The slave node can only start the wait timer when the number of successful bytes received (nob) is greater than zero as the slave node will otherwise time out even without receiving the first byte of data.

# Chapter 6

# Uplink protocol

## 6.1 Rationale

Uplink (UL) refers to direction of communication from the slave node to the master node. In wireless distributed computing, the downlink data transfer is followed by processing of the data at each slave node which is then succeeded by uplink transfer of the outputs of the processing at the slave nodes. Unlike the downlink data transfer, multiple slave nodes communicate with one master node during uplink data transfer. In such a scenario, a multiple access mechanism that can schedule transmissions from each slave node to the master node is required. Just like in case of downlink data transfer, acknowledgements for uplink data transfer also need to be scheduled to ensure reliability. So, the uplink MAC protocol like the downlink MAC protocol also needs to include a suitable channel access mechanism, reliable data transfer, rate adaptation and link failure recovery. Two different uplink protocols using two alternative channel access mechanisms have been considered for the uplink data transfer in WDC. This chapter explores the significance as well as design and implementation of these two uplink protocols.

The rest of this section briefs the other available multiple access schemes and accentuates the factors that render them to be discarded in the UL protocol.

- *Slotted ALOHA*: This belongs to the class of a random access scheme. In this access scheme, time is divided into slots. The nodes start to transmit only at the beginnings of the slots. The nodes need to be synchronized so that each node knows when a slot begins. If there is a collision of frames, the nodes detect the collision before the time slot ends. When a node has a new frame to send, it waits until the beginning of the next slot and transmits the entire frame in a slot. If there is no collision, the node has successfully transmitted the frame and retransmission is not required. If there is a collision, the node detects the collision before the end of the slot. In this case, the node transmits in the subsequent frame with a certain probability. The major drawbacks of this scheme are: i) need for time synchronization is very complex with multiple software components ii) collision detection cannot be used in half duplex systems iii) high wastage of slots due to collisions ( i.e. very low efficiency in case of multiple nodes) causing significant reduction in throughput [11].

- *Pure ALOHA*: In this multiple access scheme, no time synchronization between the nodes is required. When a node has a frame to send, it transmits it completely. If the transmitted frame undergoes collision with other transmissions, the node either immediately retransmits or otherwise, waits for frame duration. After this wait, the node transmits the frame with a certain probability. The major drawbacks of this protocol are i) collision detection cannot be supported by half duplex systems, ii) Very low utilization of resources (due to high number of collisions) [11].

- *Token passing access scheme*: In this scheme, a small special purpose frame called 'token' is exchanged among the nodes in some fixed order. When a node receives a token, it holds onto the token if it has frames to transmit, otherwise, it immediately forwards the token to the next node. If a node does have frames to transmit when it receives the token, it sends a number of frames and then forwards the token to the next node. This is highly efficient and decentralized, but the most important problem is that if one of the nodes fails, it crashes the entire network [11].

- *Time division multiplexing*: In this access scheme, time is divided into time frames which are further divided into time slots. Each time slot is assigned to a particular node. When a node has a data to send, it transmits bits from the data in its assigned time slot in the revolving TDM frame. The major drawbacks of this multiple access scheme are: i) increased implementation complexity on a software platform due to time synchronization and precise timing ii) wastage of resources (time slots) in case low number of users [11].

- *Frequency division multiplexing*: This access scheme divides a channel into different frequencies and assigns each frequency to a node in the network. The major drawbacks of this access scheme are: i) the need for band-pass filters. Since software defined radios implement these filters in software, the reliability and latency issues of software implementation pose a problem. ii) nonlinearity in power amplifiers cause the creation of out-of-band spectral components that may interfere with other FDM channels, which makes it necessary to use more complex linear amplifiers in FDM systems[11].

## 6.2 Uplink protocol I: Polling protocol

### 6.2.1 Overview

This protocol incorporates a familiar centralized multiple access technique called polling. Unlike the downlink channel access, in this scheme the master node polls each slave node in a round robin manner to request UL data as well as send acknowledgements in the reverse direction. In this protocol, a transmission from a slave node is preceded by a request from the master node. If a slave node doesn't respond to a request from the master node within a defined period of time, the master node polls the next slave node in sequence. The polling continues till entire uplink data transfer is complete. Unlike the downlink MAC protocol, this protocol polls slave nodes over multiple continuous frames in a window instead of one frame. This strategy is considered in the wake of section 8.4.2 to mitigate the effect of round trip latency constraint on throughput by reducing the number of back and forth messages. In this protocol the master node is given the ability to determine the size of a window on uplink to allow more flexibility for the optimizer block in the upper WDC layer at the master node. This protocol employs reliable data transfer, rate adaptation as well as link failure recovery. The rest of this section instantiates the overall working of this protocol and contemplates the use of such protocol.

The practical time domain representation of uplink data transfer using this protocol as obtained from a spectrum analyzer is shown in the Appendix.

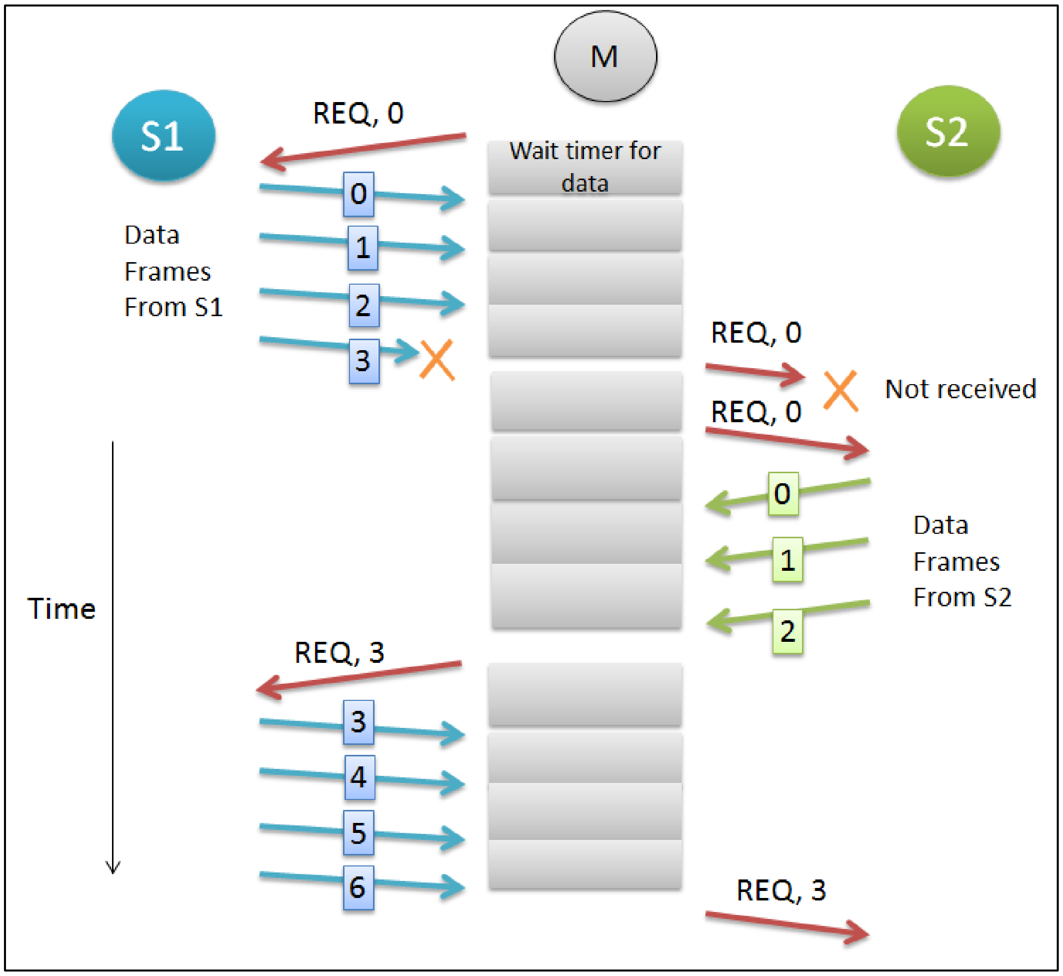


Figure 6.1: Uplink polling protocol working

The above figure illustrates UL data transfer in a WDC network consisting of two slave nodes (S1 and S2) and the master node (M). In this example, the master node chooses a window size of four for S1 and three for S2. Each uplink transmission of a window is preceded by a request for UL data (REQ) from master node. It carries the sequence number of the next expected frame from a particular slave node. In order to reduce the number of control messages exchanged, REQ is combined with a cumulative acknowledgement. In Figure 6.1, the sequence number of the frame to be requested is denoted by the number adjacent to REQ. The slave node, upon receiving the request, sends a window of frames starting from the last unacknowledged frame and then awaits its next turn. If a slave node doesn't receive a request from master node, the master node waits for an uplink data frame for a defined duration of time before sending the request again.

The major advantages of using polling multiple access scheme are:

i) Resources (Bandwidth, power, etc.) are utilized efficiently as the chances of collisions between two transmissions are reduced owing to the turn-based approach. It also eliminates empty slots.

ii) It eliminates the hidden node problem as two slave nodes cannot transmit at the same time in such a protocol.

The disadvantages of this access scheme are: (i) polling delay and (ii) centralized nature. In order to overshadow the effects of polling delay greatly influenced by round trip latency, multiple frames are transmitted in a window continuously. Consequently the data arrives in bursts of high data rates. The centralized nature of this access scheme is not really a problem in the realm of WDC as it embraces a centralized architecture.

### 6.2.2 Design and Implementation

This section uses the naming conventions or symbolic variables listed in Tables 6.1 and 6.2 to describe the working and implementation of the UL MAC protocol.

Table 6.1: Naming conventions used in Uplink polling protocol at the Master node

| | |
|---|---|
| ping_slv_count[k] | The successive number of times the master node has requested the k$^{th}$ slave node for UL data during a particular poll. |
| ping_slv_limit | The defined maximum number of times the master node requests a slave node for UL data in a poll before polling the next slave node. |
| time stamp | A number uniquely identifying a request message for UL data. |
| node_served | The current slave node polled. |
| max_UL_attempts | The defined maximum number of consecutive polls the master node requested a slave node but did not receive any uplink data. |
| send_win$_{SL}$[k] | The number of frames requested by the master node from the k$^{th}$ slave node during a poll. |

Table 6.2: Naming conventions used in Uplink polling protocol at a slave node

| | |
|---|---|
| rcv_pid | Sequence number in the acknowledgment or UL data request. |
| nexp | Sequence number of the next expected frame to be sent to the master node. |
| $\text{win\_count}_{SL}$ | Number of frames successfully received by the master node from the previous window. |
| p_len_array | An array that stores the data message length of each frame sent in the previous. |
| p_sent_array | An array that stores the sequence numbers of the frames sent in the previous window. |

### 6.2.2.1 Request for UL data

A slave node has no knowledge of when the overall downlink data transfer completes. The request for UL data initiates the UL data transfer process by notifying a slave node that the master node is ready to receive. It makes the job of a slave node simpler. If the slave node is ready to send UL data, it replies to the request with a window of data frames. In case the slave node is in the processing phase following the DL data transfer, it simply responds to the master node indicating it is still busy so that the master node can poll the next slave node in sequence.

The master node takes the following steps while requesting *node_served* for uplink data:

i. If the uplink data transfer from *node_served* completes with the previous uplink window, the master node sends a final acknowledgement to *node_served* to enable termination of the uplink data transfer. Once a final acknowledgement has been sent, the master node adds *node_served* to a list of slave nodes that will not be polled. The next time the master node needs to poll the same slave node, it skips to the next slave node in sequence which has data to send. When the master node polls another slave node, it repeats step i. If however, *node_served* has data to send, the master node executes step ii. The master node is able to determine whether a particular slave node has data to send because it obtains information regarding total number of bytes to be transferred from the data frame header.

ii. During a particular poll, the master node can send a maximum of *ping_slv_limit* number of UL data requests. Each UL data request is followed by its wait period. After every request the counter *ping_slv_count[node_served]* is incremented by one. If the master node doesn't receive UL data in a poll, *UL_attempts[node_served]*is incremented by one. Both these counters are reset on receiving data from *node_served*.

    So, the following conditions must be satisfied to allow the master node request *node_served* for UL data:

    $$\text{ping\_slv\_count[node\_served]} \leq \text{ping\_slv\_limit}$$

    $$\text{UL\_attempts[node\_served]} \leq \text{max\_UL\_attempts}$$

    If these conditions are not met, the master node simply polls the next slave node in sequence and repeats step 1.

Each request for UL data contains a time stamp. A session basically refers to the period of time during which the master node requests a slave node for uplink data. The value of time stamp is incremented only once every such session as depicted in Figure 6.2. This figure shows a slave node will only accept an UL data request from the master node if the value of the time stamp in the message is different from the value of the last time stamp received. This is a measure to reduce redundancy by preventing response to duplicate uplink data requests in the same session. This can occur in a situation when more than one UL data request from the same session arrives in the buffer at the slave node when it does not get flushed. Figure 6.2 also represents how a request for UL data also acts as a cumulative acknowledgement. The request for UL data message also carries information regarding the size of the window and information about frame errors or frame drops at the master node.

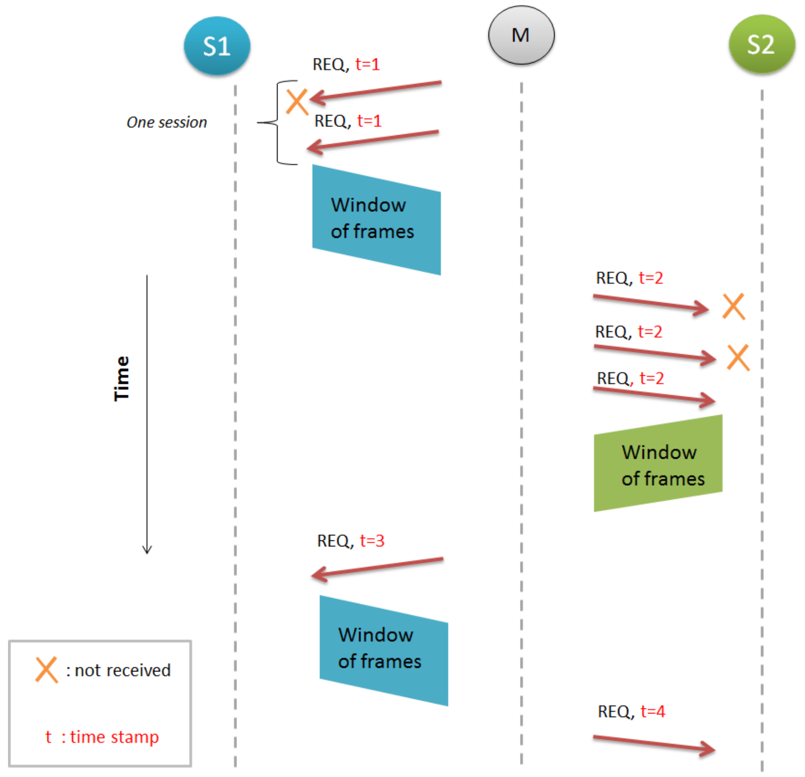


Figure 6.2: Time stamps in Uplink protocol

### 6.2.2.2 UL data transfer

A slave node essentially goes through three major phases in WDC, the DL data transfer phase, processing phase and UL data transfer phase. The uplink data transfer phase at a slave node consists of *Receive UL data request mode* and *send data mode*. When a slave node receives an UL data request from the master node, it switches from the default *RX mode* to *TX mode* to send uplink data. The UL data request carries the sequence number of the first frame in a window. Just like the downlink protocol, the uplink protocol also employs reliable data transfer mechanism which consists of a sliding window mechanism and a data continuity mechanism.

The sliding window mechanism[11] helps in maintaining the correct sequence of data frames while the data continuity mechanism ensures continuity in data across the frames.

The following transformations take place at the slave node in *send data mode* after every frame is pushed out of the MAC component.

- Sequence number of the next frame to be sent is updated

  $\text{nexp} = (\text{nexp}+1)$ AND $0\text{xff}_{\text{HEX}}$.
- Number of frames transmitted from the current window is updated

  $\text{win\_count}_{\text{SL}}=\text{win\_count}_{\text{SL}}+1$.
- The sequence number of the most recently sent frame is stored in *last_p_sent_array*.
- The data size of most recently sent frame is stored in *last_p_len_array*.

After a whole window has been sent, the slave node switches back to *RX mode* to and waits for the acknowledgement or UL data request from the master node. This can also happen if the slave node has no more data left to send. As the lengths of a window can be changed by the rate adaptation algorithm, the arrays *last_p_len_array* and *last_p_sent_array* are created at the beginning of every window to accommodate the next window size. When the slave node receives an UL data request from the master node, if *rcv_pid* does not match the value of *nexp*, it implies that the master node didn't receive the most recent window successfully. The slave node slides its window such that sequence number of the first frame of the next window is *rcv_pid*. The unacknowledged frames must be retransmitted. Prior to the retransmission, the slave node must ensure that the file pointer is located at the right location in the file so that no discontinuity occurs in the retransmitted data. The data continuity mechanism is implemented a little differently from the downlink protocol even though the idea is the same. According to the data continuity mechanism, once a slave node receives an acknowledgement asking for retransmission, it looks for the sequence number *rcv_pid* in the *last_p_sent_array*. Every time the master node accesses an element in *last_p_sent_array*, the file pointer in the data file is moved in the reverse direction by the data size of the corresponding frame given by the same array index location in *last_p_len_array*.

Each slave node can send the same window for defined $\textit{same\_win\_count}_{max}$ number of times before terminating the data transfer to the master node. Each slave node also maintains a counter for the number of bytes successfully received by the master node. When this number equals the size of the file being transferred data transfer on that link is complete. When the master node receives all the data successfully from a particular slave node, it doesn't poll that node any further.

Figures 6.3 and 6.4 illustrate with the help of an example how the *last_p_len_array* and *last_p_sent_array* are filled and accessed in the uplink protocol. When *last_p_len_array* is created at the beginning of every new window, all elements of that array are initialized to zero, while all elements of *last_p_sent_array* are initialized to a value of greater than 255 (larger than the largest possible sequence number, 255).

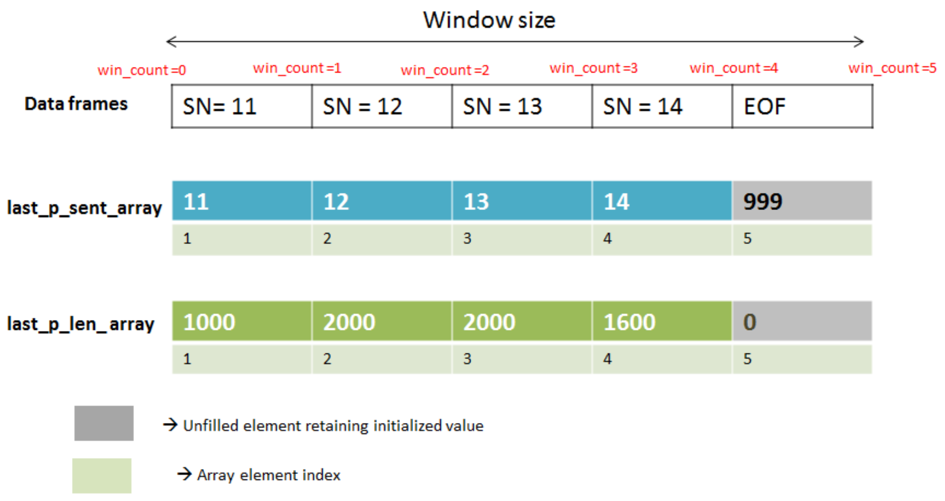


Figure 6.3: Storing frame information during UL data transfer

The above figure demonstrates how *last_p_len_array* and *last_p_sent_array* are filled while transmitting a frame with a given sequence number (SN) in a window. EOF represents that there is no more data left to send; in this case it occurs after frame with sequence number 14. The unused elements of *last_p_len_array* and *last_p_sent_array* retain their initialization values.

Figure 6.4 illustrates how the *last_p_len_array* and *last_p_sent_array* are accessed prior to a retransmission.

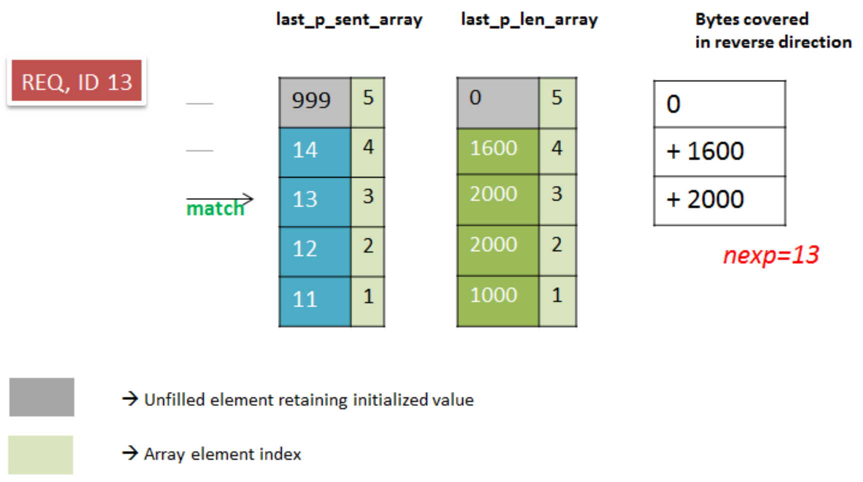


Figure 6.4: Retrieving frame information during UL data transfer

In this example, frame with sequence number 13 needs to be retransmitted. Thee slave node starts searching the *last_p_sent_array* for sequence number 13. For every element accessed in the *last_p_sent_array*, the file pointer is moved in the reverse direction by the number of data bytes located at the same index in the *last_p_len_array*. This continues till sequence number 13 is found. In this example, the file pointer is moved 3600 bytes in the reverse direction before the next window is transmitted with the sequence number of the first frame as 13. It must be noted that unlike the downlink protocol, the slave node does not need to maintain state of the previous window sent.

### 6.2.2.3 State machine

The master node plays a pivotal role in UL data transfer. It receives data from multiple slave nodes sends acknowledgements to each of them. Figure 6.5 represents the finite state machine of the uplink protocol at the master node. A finite state machine is simply a way to describe an algorithm. The event causing the transition is shown above the horizontal line labeling the transition and the actions taken when the event occurs are shown below the horizontal line. '&&' represents a logical AND while '||' represents a logical OR.

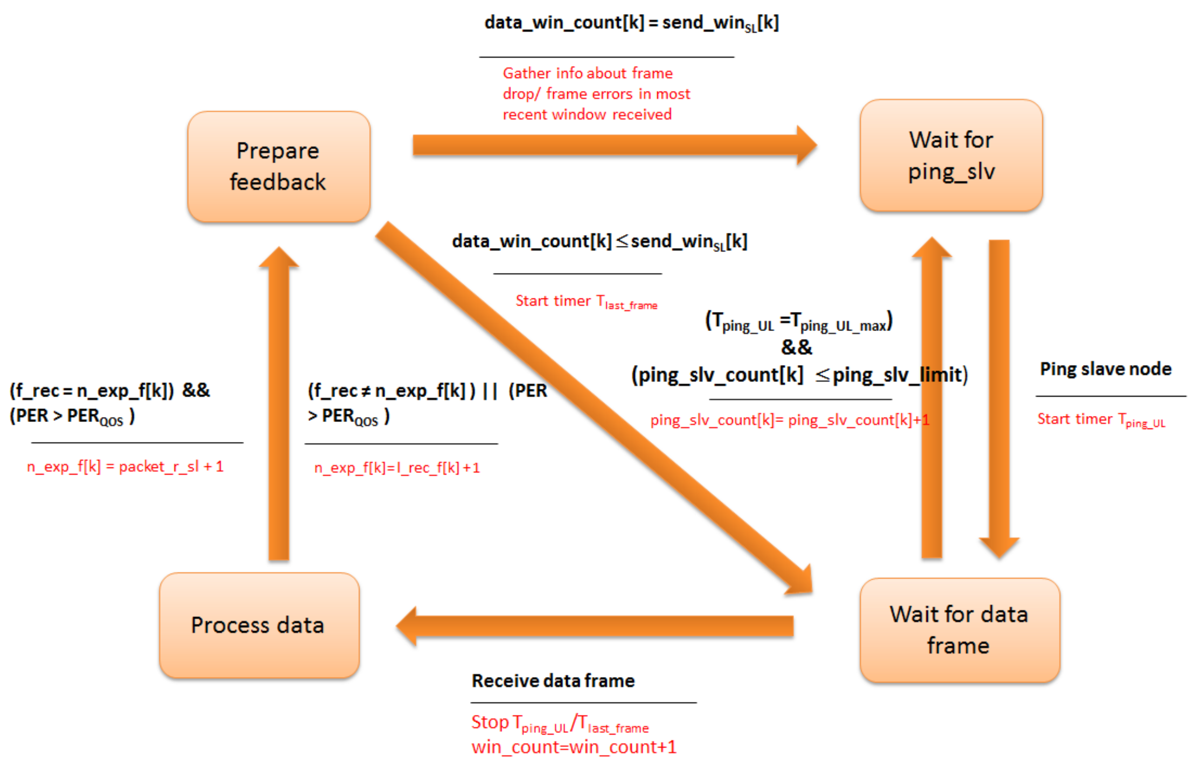

data_win_count[k] = send_win_SL[k]
Gather info about frame drop/ frame errors in most recent window received
Prepare feedback
Wait for ping_slv
data_win_count[k] ≤ send_win_SL[k]
Start timer T_last_frame
(T_ping_UL = T_ping_UL_max) && (ping_slv_count[k] ≤ ping_slv_limit)
ping_slv_count[k]= ping_slv_count[k]+1
Ping slave node
Start timer T_ping_UL
(f_rec = n_exp_f[k]) && (PER > PER_QOS )
n_exp_f[k] = packet_r_sl + 1
(f_rec ≠ n_exp_f[k] ) || (PER > PER_QOS )
n_exp_f[k]=l_rec_f[k]+1
Process data
Wait for data frame
Receive data frame
Stop T_ping_UL/T_last_frame
win_count=win_count+1

Figure 6.5: State machine at the Master node in UL Polling protocol

In the above state machine, there are four major states represented by the boxes: waiting for a command to send UL data request, waiting for data frame, processing received data and reliable and continuous data transfer. The system starts with the master node waiting for *ping_slv* command (from an upper layer) in the *Wait for ping_slv* state which basically triggers the UL data request (or acknowledgement) for a slave node. Once such a command is received, the master node sends an UL data request to a slave node. As soon as it sends this request, it starts a timer $T_{ping\_UL}$ and waits for a data frame from the slave node. If the master node doesn't receive a data frame within the duration $T_{ping\_UL\_max}$, it retransmits the UL data request assuming all the necessary conditions are satisfied (as explained in 6.2.2.1). Once a data frame is received, it is processed. As soon as a data frame is received, another timer $T_{last\_frame}$ is started as the master node simultaneously moves into the *Process data* state. In this state the master node decodes the actual data and stores it in an output file.  In the *Prepare feedback* state, the master node updates all the header elements (frame drops, frame errors, time stamp, etc.) for the acknowledgement frame and puts the next expected sequence number.

The following figure summarizes the major steps in the uplink data transfer as discussed throughout section 6.2.2.

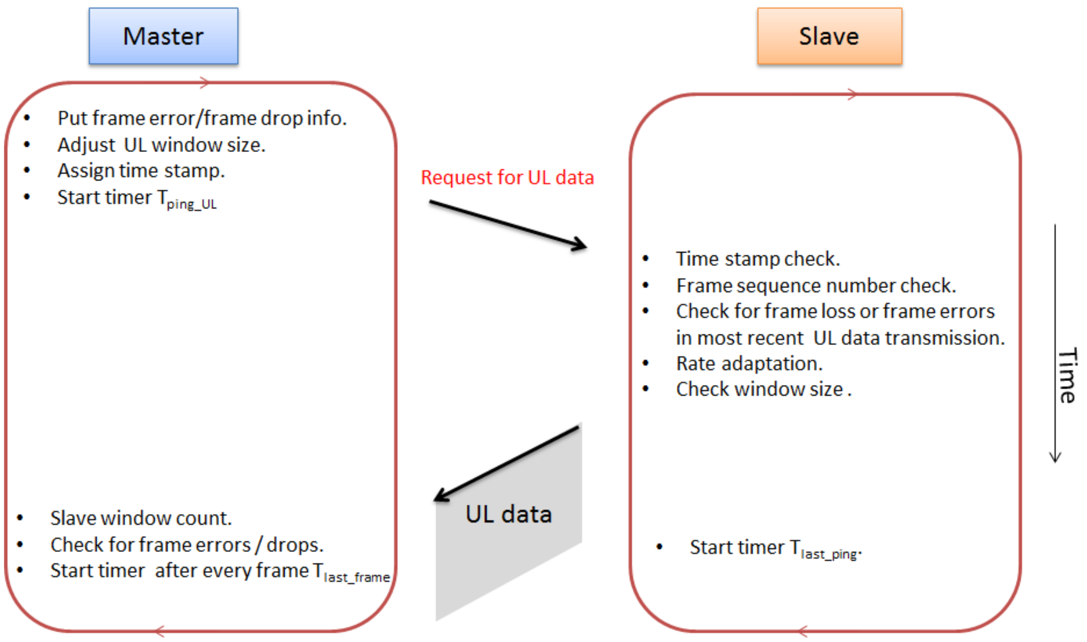

Master
Slave
• Put frame error/frame drop info.
• Adjust UL window size.
• Assign time stamp.
• Start timer $T_{ping\_UL}$
Request for UL data
• Time stamp check.
• Frame sequence number check.
• Check for frame loss or frame errors in most recent UL data transmission.
• Rate adaptation.
• Check window size .
Time
UL data
• Slave window count.
• Check for frame errors / drops.
• Start timer after every frame $T_{last\_frame}$
• Start timer $T_{last\_ping}$.

Figure 6.6: Steps involved in UL data transfer

### 6.2.3 Link failure

A link failure means a node cannot communicate with the other either due to sever channel conditions or some problem in hardware/software. Channel conditions on each link can be different from the other and can vary with time. The use of timers and counters make the overall protocol more robust. In case of a short term link failure, the slave node may not receive an uplink data request or acknowledgment from the master node. This is where the timer $T_{ping\_UL}$ plays an important role. Each time the master node sends out an uplink data request (or acknowledgement), the $T_{ping\_UL}$ timer is started. As already mentioned, the master node can send a defined *ping_slv_limit* number of requests in one session. This means the master node waits for a $T_{ping\_UL\_max}$ duration before polling the next slave node.

$$T_{ping\_UL\_max} = T_{ping\_UL} \ . \ ping\_slv\_limit$$

The slave node can send the same window for *max_UL_attempts* times before terminating data transfer on that link. This prevents looping and redundancy in retransmissions in case the channel conditions become severe for a long period of time. After every frame received (successfully or unsuccessfully), the master node starts a timer $T_{last\_frame}$ and waits for $T_{last\_frame\_max}$ duration before polling the next slave node.

The slave node also simultaneously uses a timer to makes the uplink protocol robust to link failures. At the end of every window, a timer $T_{last\_ping}$ is started at the slave node. If the slave node doesn't receive any acknowledgement or UL data request within duration $T_{last\_ping\_max}$, it terminates the UL data transfer on that link. A suitable choice for $T_{last\_ping\_max}$ is:

$T_{last\_ping\_max} > ((N_{sl}-1) . (T_{ping\_UL} .$ ping_slv_limit)) + (send_win$_{SL}$[k] . (Transmission delay of frame) . $(N_{sl}-1))$ ; $N_{sl}$ is the number of slave nodes in the network.

## 6.3 UL protocol II: CSMA/CA uplink protocol

### 6.3.1 Overview

One drawback of the UL polling protocol is the centralized nature i.e. the master node has to coordinate the channel access of the different slave nodes. It would certainly take the burden off the master node if the slave nodes can schedule among themselves access the channel without collisions. This is the motivation behind using an alternative uplink protocol CSMA/CA on the uplink. It uses a multiple access scheme, CSMA/CA, which is a random channel access scheme and a counterpart of CSMA/CD (carrier sense multiple access with collision detection) used in wired LANs.

CSMA/CD is not suitable for wireless networks mainly because of the following reasons:

- Implementing a collision detection mechanism requires the implementation of a Full Duplex system (i.e. ability to detect the channel and transmit data simultaneously) [11].

- In a wireless environment, all nodes may not be able to hear each other due to hidden node problem. The use of CSMA/CD would result in high number of collisions [11].

In CSMA/CA, a node listens to the channel before transmission to determine whether someone else is transmitting.  The receiving node sends an acknowledgement after receiving the data. If an acknowledgement is not received retransmission occurs. This scheme uses control messages like RTS (Request to send) and CTS (clear to send) to eliminate the hidden node problem. A sending node senses the medium for a specified time, DIFS, (Distributed Inter frame space , as called in IEEE 802.11 standard) and then transmits a control message RTS to the destination. The destination will respond to the RTS with CTS. All other nodes receiving either an RTS and/or CTS defer access to the channel medium for a certain period of time which denotes the time for which the channel has been reserved. This is called virtual carrier sensing. This reduces the probability of collisions at the receiver by a node which is hidden from the transmitting node. RTS and CTS are short messages and reduce the overhead of collisions. If a channel is sensed as busy (either another node transmission or interference) during DIFS, the node waits the end of the current transmission and then starts the contention (wait a random amount of time). When its contention timer expires, if the channel is still idle, the node sends the packet. The node having chosen the shortest contention delay wins and transmits its packet. The other nodes just wait for the next contention (at the end of this packet). The contention is a random number and occurs for every data frame [11, 19].

## 6.3.2 Design and Implementation

In the context of WDC, CSMA/CA algorithm is implemented only on the uplink because multiple salve nodes are communicating with a single master node at possibly the same time.

The algorithm consists of the following rules.

- Prior to any transmission, a slave node senses the channel for a predefined DIFS duration. If it does not sense any energy (energy above the squelch threshold) in the channel, it interprets the channel as idle and sends the RTS message.

- If the slave node senses any energy in the channel, it defers channel access for $T_{DEFER}$ duration of time so that the channel is reserved by an active pair of nodes (one slave node and the master node).
- The master node replies to an RTS from a slave node with corresponding CTS.
  It maintains an inactive period $T_{im}$ after sending out CTS to a particular slave node, say X, during which it discards any RTS from any slave node but only accepts data message from that particular slave node X. This facilitates virtual carrier sensing and channel reservation from the master node side. Without this mechanism, it has been observed that the multiple channel access doesn't work on this system.
- If a slave node doesn't receive the corresponding CTS to an RTS within duration of $T_{CTS}$, the slave node enters a contention period.
- The contention period is always followed by DIFS during which the slave node senses the channel. A slave node goes into contention after defer access mode, after $T_{CTS}$ timeout and after sending out a data message.

In traditional CSMA/CA implemented in the 802.11 standard, a receiving node sends an acknowledgement after receiving data. In this protocol, the acknowledgement is combined with the CTS message. In other words, the acknowledgement for a packet comes in the next arriving CTS message. The advantage of this mechanism can be explained with an example. Assume that node A sends N packets of the same size to node B. The length of every control message (RTS, CTS, ACK) is typically the same, say $T_{ctrl}$. The length of any message in time domain is equivalent to its transmission delay.

Neglecting times spent in DIFS and contention period, as per traditional CSMA/CA,
Total time taken in the data transfer = N. (RTS+CTS +DATA+ACK) = $3NT_{ctrl}$ + N.DATA

Using this scheme,
Total time taken in data transfer=N.(RTS+CTS+DATA)+(RTS+CTS) = (2N+2) $T_{ctrl}$ + N.DATA

Besides taking lesser time in the data transfer, lesser number of control messages are exchanged which reduces the overhead.
Every slave node uses the same contention window during the contention period. This is because in a WDC network, each slave node during the DL data transfer overhears the transmissions

from the other slave nodes and calculates the total number of slave nodes in the network. Since the number of slave nodes in the network is known, the exponential backoff in the contention period is not required.

The following figure demonstrates the working of CSMA/CA as implemented on OSSIE

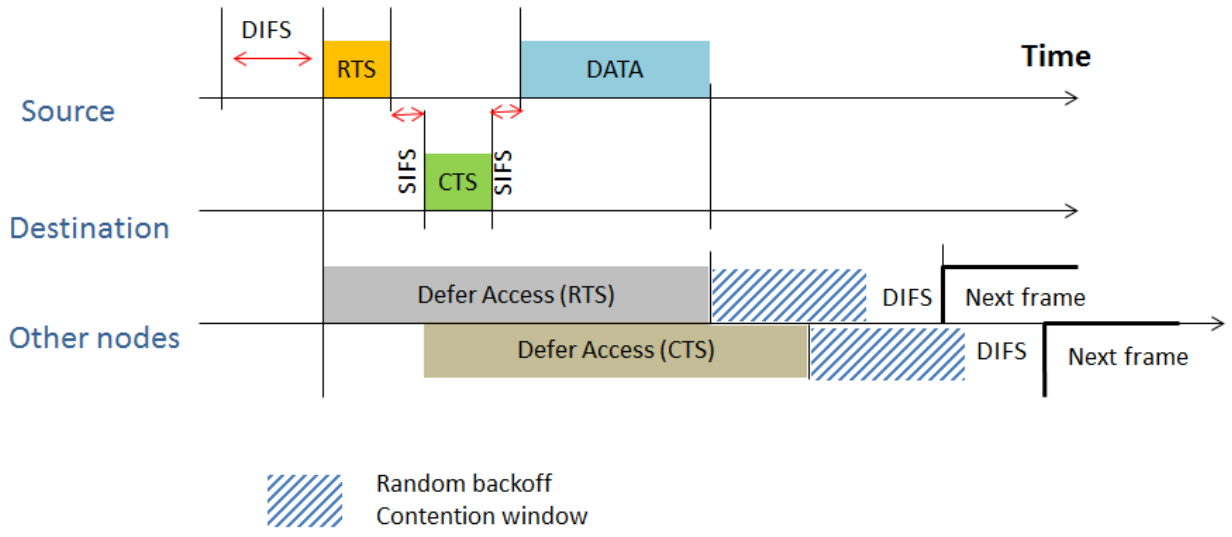


Figure 6.7: CSMA/CA Uplink protocol working

In the above figure, SIFS stands for short inter frame space which denotes the time taken by a node to switch from receive to transmit mode or vice versa. A source node (slave node) senses the channel idle for DIFS period of time and sends an RTS to the destination (master node). Other nodes which are sensing the channel find the channel busy and defer channel access for $T_{DEFER}$ period of time. The master node responds to the RTS with CTS. When the source node receives the CTS, it sends a data frame to the master node. When the other slave nodes recover from their respective 'defer access' times, they go into a contention period. The source node also goes into the contention period. The slave node with the shortest contention period gets to sense the channel and if it is idle for DIFS duration, it sends a data frame. The random value of contention period allows different slave nodes to send data at different times.

All the timers described in this section consider the fact that propagation delay and processing delay are minuscule compared to the transmission delay and system latency. From figure 6.7, the length of defer access mode duration is chosen as:

$$T_{DEFER} > t_{RTS} + t_{CTS} + t_{DATA}$$

$t_{RTS}$, $t_{CTS}$ and $t_{DATA}$ are the lengths of RTS, CTS and data frame in time domain.

$$t_{RTS} = t_{CTS} = T_{ctrl} \Rightarrow T_{DEFER} > 2T_{ctrl} + t_{DATA}$$

The size of a control message is constant; so $T_{ctrl}$ is constant. The length of the data message can vary but the rate adaptation algorithm adjusts the length of the message such that the number of samples pushed into the USRP[2] is the same for every frame. As a result of this, $t_{DATA}$ is constant. Hence, $T_{DEFER}$ is a constant.

The wait time for CTS can be expressed as:

$$T_{CTS} > 2T_{ctrl}$$

In practice the duration of all the wait times is assigned a large enough value to accommodate the round trip latency in the system. The practical time domain representation of uplink data transfer using CSMA/CA protocol as obtained from a spectrum analyzer is shown in the Appendix.

## 6.3.3 Implementation challenges

The most significant challenges to implement CSMA/CA on OSSIE are listed below-

- The interface between the slave_node_v2 component and *FlexframeSync* component in the OSSIE waveform consists of two ports: one for the header and the other for the payload. The *FlexframeSync* component senses energy in the channel prior to checking for a valid physical layer header. The channel is considered busy if and only the header is valid. In case of channel sensing, this channel status needs to be relayed to the subsequent MAC component. This information is relayed through a special interface (shared user space), which brings up stability issues pertaining to concurrent memory access by multiple processes.

- An OSSIE component calls a function (*pushPacket*) to push the data out of the output port. Each component in a waveform uses this function to push data out of its respective output port. It has been shown in section 8.3 that there is an average latency of 5ms on the transmit chain on one node from the time data is pushed out of the MAC component to the time data actually comes out of the USRP. Also, the round trip latency varies. This poses a big problem in carrier sensing because by the time a slave node realizes the channel is idle and sends an RTS, the channel may actually be occupied.

- When a *FlexframeSync* component receives a valid header, it does not know the state of the subsequent *slave_node_v2* component (i.e. whether it is in channel sensing mode during DIFS or waiting for a CTS or data) in the receive chain. In case the MAC component is in channel sensing mode, the payload of the MAC frame should be discarded while at other times it should decode the information in the payload. This can be realized with the help of a reverse connection (from *slave_node_v2* component to *FlexframeSync* component). This poses a daunting challenge of ensuring continuous flow of information across the entire waveform at all times.

# Chapter 7

# Rate adaptation

## 7.1 H-ARQ (Hybrid automatic repeat request)

It is a widely known method which uses a combination of forward error correction bits and error detection (or ED) bits to ensure data delivery with reduced errors. In the standard ARQ (automatic repeat request), only error detection bits are used which help to detect the errors in the received frames while in H-ARQ, the receiver attempts to correct the received bits and then verifies whether there are any errors in the received data. H-ARQ reduces the number of retransmissions compared to ARQ and therefore boosts the throughput in bad channel conditions but it also reduces the efficiency of the system in good channel conditions[20].

There are two major types of HARQ schemes – Type I and Type II. In Type 1, both ED and FEC redundant bits are added to each message prior to transmission. When the coded frame is received, the receiver first decodes the error-correction code and then uses the error detection bits to check if all the errors have been corrected. If the number of erroneous bits is within the QOS requirement, retransmission is avoided. In Type II HARQ, the source performs incremental Redundancy to transmit additional redundant information in each retransmission and receiver decodes each retransmission. Error detection typically only adds a couple bytes to a message, which is only an incremental increase in length. FEC, on the other hand, can often double or triple the message length with error correction parities[20].

The MAC protocol for WDC incorporates a variation of Type II HARQ. We will look into the implementation of the HARQ for now. In this section the terms 'packet' and 'frame' are explicitly different. A frame is simply an encapsulation of smaller packets. Each packet has its own cyclic redundancy check (CRC) bits for error detection.

A brief description of the implementation of CRC check is as follows:

Consider d-bits of data to be sent, D. The sender and receiver must agree on a (r+1) bit pattern known as the generator 'G', where r is the number of redundant bits. The most significant bit of G is 1. For a given piece of data, the sender will choose the r additional bits 'R' and append them to D such that the resulting d+r bit pattern is exactly divisible by G using modulo 2 arithmetic. If the remainder is not zero, data has an error.

R = remainder $((D.2^r)/G)$, 2r means bit shift towards left r places.

This MAC layer uses a 32 bit CRC, G = 100000100110000010001110110110111 [11].

The following figure shows the structure of the payload part of the MAC frame.

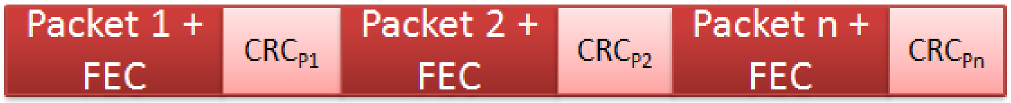


Figure 7.1: Packet structure

The above figure depicts that each packet has its own set of redundant CRC bits. On the transmitter side, error detection bits for each packet are determined, then the packet is coded using a forward error correction scheme and finally the ED bits are added to the end of an encoded packet. On the receiver side, each encoded packet is first decoded and then the corresponding CRC bits are used to check for error in the packet. The receiving node calculates the packet error rate or PER for every frame. If the PER is within the defined $QOS_{PER}$ threshold, a retransmission of the frame is avoided.

The FEC codes used in this MAC protocol are convolution codes supported by the 'lib-fec' library in C programming language [21].

But how does the receiving side know of the size of the encoded packet? We know that the length of the packet depends on both the original length of the message and the code rate of a

FEC code. This poses an implementation challenge because in practice, a code rate of (x/y) of a FEC code does not necessarily mean that the number of redundant bits is (y-x). It actually refers to the asymptotic rate which basically means that if the number of bits is really large, the code rate converges to (x/y). In order to solve this problem, the original size of a packet for a particular frame is notified by the sending node through the MAC header. We have already learnt that the MAC header also carries information about the FEC scheme (type and code rate) used on the data. The receiving node harnesses the information about the original size of a packet and the FEC scheme to calculate the size of an encoded packet in the received frame. It already knows the number of error detection bits, which is constant.

From implementation perspective, the receiving node needs to create an array for the decoded message to be stored. The size of the decoded message can be accurately determined only when the number of packets contained in the frame is known. This is calculated by using the following expressions:

$$\textit{Number of packets in a frame} = \frac{\textit{Payload length}}{\textit{Encoded packet size}}$$

$$\textit{Size of actual data in a frame} = \textit{Original packet length} \;.\; \textit{Number of packets in a frame}.$$

This is a good time to explain the use of the header element ‘multiple’. We know that the MAC header is defined to be eight bytes long. To make sure that the size of a packet is conveyed using at most 8 bits, the packet size is expressed as an integer multiple of a predefined number. This allows us to use the limited header size resourcefully.

## 7.2 Rate adaptation algorithm

As previously mentioned, this MAC protocol uses a variation of type II HARQ which uses incremental redundancy. When the number of redundant bits is changed, the goodput data rate will definitely be affected. But how will this redundancy be controlled? This question is answered by the rate adaptation algorithm. Rate adaptation means that with varying channel conditions, the system is able to change certain parameters which have direct effect on the instantaneous data transmission rate. This makes the system more robust and reliable as it enables the MAC protocol to perform under varying channel conditions. In this section, we will understand the rate adaptation algorithm designed for this MAC protocol, parameters concerned with this rate adaptation scheme and then analyze its performance.

The rate adaptation algorithm uses several coding schemes which have different code rates, constraint lengths and minimum free distances. As the free distance increases, the performance of the convolutional code increases. Table 7.1 lists the convolutional code schemes considered in this MAC protocol:

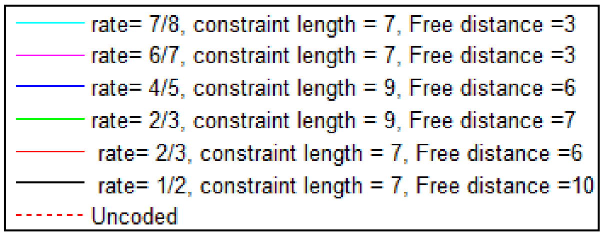
Table 7.1: Convolutional codes

| Asymptotic code rate | Constraint length | Free Distance |
|---|---|---|
| 7/8 | 7 | 3 |
| 6/7 | 7 | 3 |
| 4/5 | 9 | 6 |
| 2/3 | 9 | 7 |
| 2/3 | 7 | 6 |
| ½ | 7 | 10 |

The goal of the rate adaptation algorithm is to be able to push as many message bits as possible at the highest rate such that the number of erroneous bits (or rather packets) the receiver receives is within the QOS threshold. If a transmitting node adapts based on the SNR received at the receiver, this will mean that the sender will use a very limited number of combinations of modulation scheme and FEC scheme to adapt. The problem that arises in such a method of rate adaptation is that even if the average received signal is within a given range, instantaneous fluctuations due to loss of line of sight (moving obstacles) can easily cause the instantaneous received signal to go below the lower bound of the SNR range within which a combination of FEC scheme and modulation scheme works best. This will cause a frame error resulting in all the subsequent packets in the send window to be discarded at the receiver. In such a case, the sender will retransmit frames with the same combination of FEC and modulation scheme only to result in a similar outcome. So, we can see that the more important thing to consider is whether the frames sent out in the last window has been received by the receiver or not. This is the motivation for the following rate adaptation algorithm.

Each of the following set of graphs represents the theoretical PER obtained using different FEC schemes for a given modulation scheme at different channel conditions. These graphs have been obtained by running simulations on a laptop using an additive white Gaussian Noise channel. Packets of size 128 bytes are sent and received over the simulated channel till the maximum number of bits have been pushed to achieve the lower bound of the packet error rate. The SNR range of operation has been defined to be 4dB to 14dB.

Legend for the graphs in Figures 7.2-7.5:



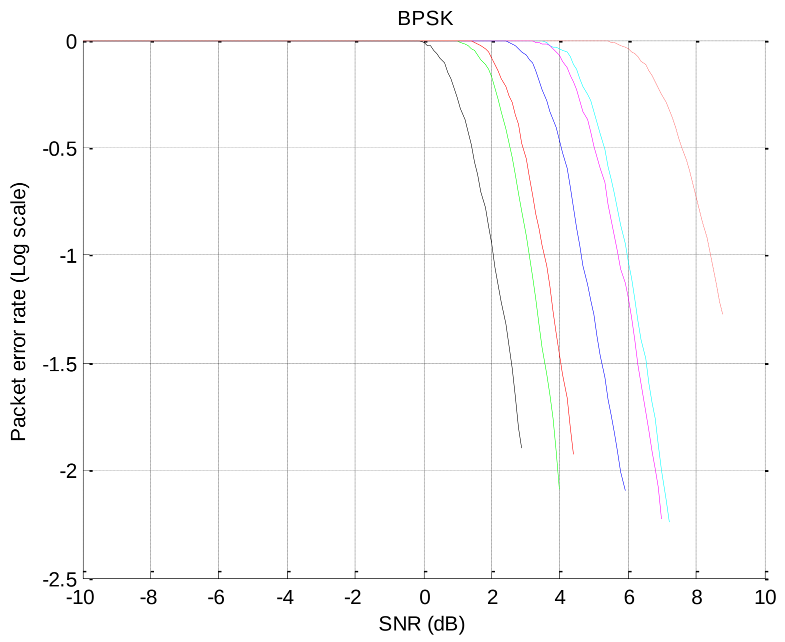


Figure 7.2: Theoretical Packet error rate vs. SNR for convolutional codes (BPSK)

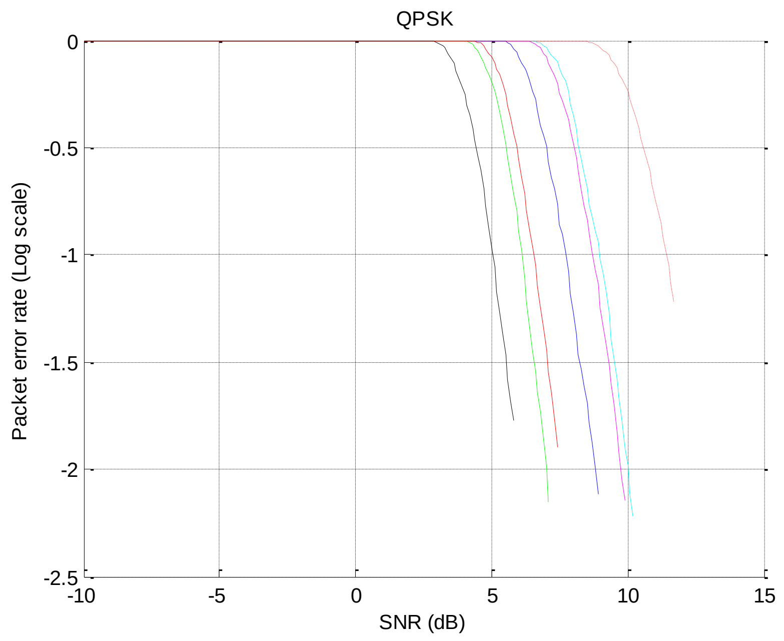


Figure 7.3: Theoretical Packet error rate vs. SNR for convolutional codes (QPSK)

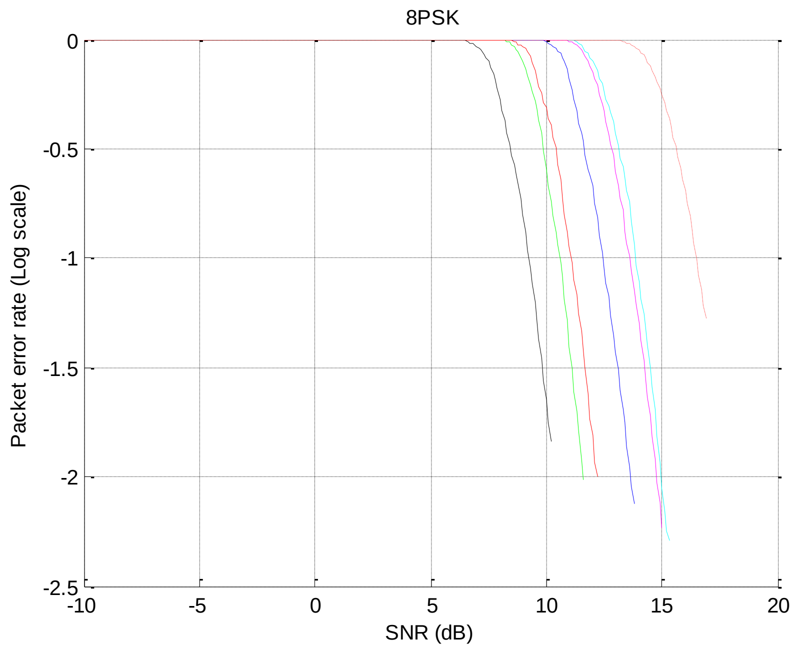


Figure 7.4: Theoretical Packet error rate vs. SNR for convolutional codes (8-PSK)

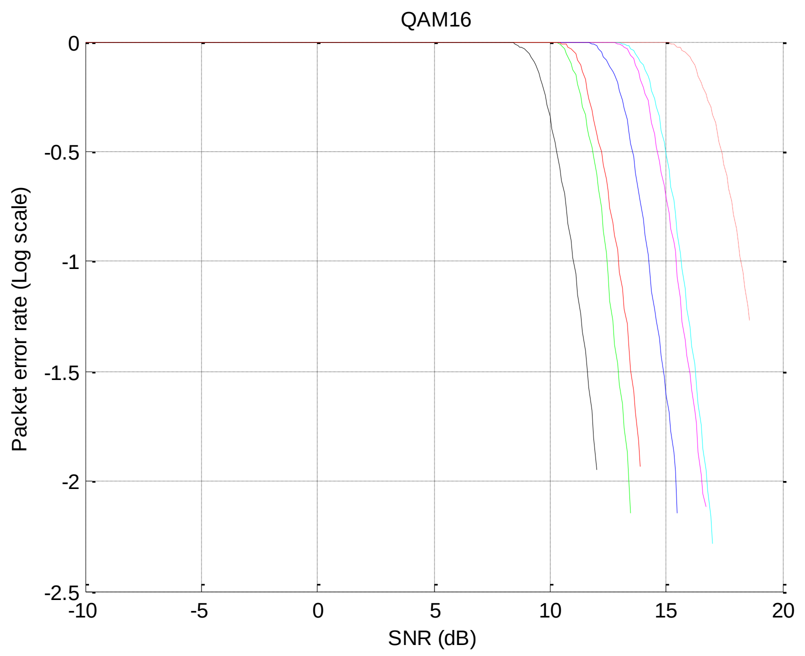


Figure 7.5: Theoretical Packet error rate vs. SNR for convolutional codes (16-QAM)

Figures 7.2-7.5 show how performance varies by switching the FEC schemes. It is intuitive that the convolutional code with rate ½, constraint length =7 and free distance=10 is the strongest FEC scheme as well as the most inefficient. Whereas, the convolutional code with code rate 7/8, constraint length = 7 and free distance= 3 is the weakest but most efficient one. While using one modulation scheme, a node can gradually switch encoding schemes to increase or decrease efficiency such that the data transmitted is received error free at the receiver. This is what the rate adaptation algorithm exploits.

Let us look at how the modulation scheme and FEC scheme affect the size of the frame. The maximum number of samples in a frame has been set to 22000. Let length of the original data message in the frame be L bytes = 8L bits.

FEC code rate = (x/y); for x useful bits (y-x) redundant bits.

$\therefore$ Size of the frame after FEC = $\left\{ Ceil\left(\frac{8L}{x}\right)(y-x) \right\} + 8L$ bits

Number of bits per symbol (depending upon the modulation scheme) = N

Interpolation rate = I samples/symbol; fixed at 4

$\therefore$ Total number of samples pushed to the USRP = $\left\lceil \frac{\left\{ Ceil\left(\frac{8L}{x}\right)(y-x) \right\} + 8L}{N} \right\rceil I \quad \le 22000$

So, $L = Ceil\left(\frac{22000Nx}{8Iy}\right)$ bytes. It is also to be noted that L needs to be an integer multiple of the packet size (unless it is the very last packet) as mentioned previously.

The variables used to describe this rate adaptation algorithm are listed in Table 7.2 below.

Table 7.2: Naming conventions used in Rate adaptation algorithm

| | |
|---|---|
| next_fid | Sequence number of the next frame expected to be sent. |
| l_win_beg | Sequence number of the first frame in the most recently sent window. |
| l_win_size | Sequence number of next frame expected to be sent. |
| same_frame_count | Successive number of times the same window (denoted by the frame ID of the first frame in a window) is sent. |
| ack_p_drop | A flag, when set, denotes that at least one frame from the most recent window was dropped at the receiving node. |
| ack_ber_bit | A flag, when set, denotes that at least one frame from the most recent window has been discarded at the receiver because of errors. |
| packet_id_T | Sequence number of the first frame of the next window to be sent. |
| last_ack_rec | A flag, when set, denotes that the acknowledgement was received for the most recent window sent. Otherwise, it means there has been a timeout. |

Tables 7.3 and 7.4 list the modulation scheme indices and FEC scheme indices used in the rate adaptation algorithm.

Table 7.3: Modulation scheme index

| Modulation | Index (MI) |
|---|---|
| BPSK | 1 |
| QPSK | 2 |
| 8PSK | 3 |
| 16QAM | 4 |

Table 7.4: FEC index

| FEC scheme | Index (FI) |
|---|---|
| None | 0 |
| CONV, $r=7/8$,$K= 7$, $d_f=3$ | 1 |
| CONV, $r=6/7$,$K= 7$, $d_f=3$ | 2 |
| CONV, $r=4/5$,$K=9$,$d_f=3$ | 3 |
| CONV, $r=2/3$,$K=7$, $d_f=3$ | 4 |
| CONV, $r=2/3$,$K=9$, $d_f=3$ | 5 |
| CONV, $r=1/2$, $K=7$,$d_f=3$ | 6 |
| CONV, $r=1/3$, $K=9$, $d_f=3$ | 7 |

r = code rate, $d_f$ = free distance, K= constraint length, CONV = convolutional code

Figure 7.6 describes the rate adaptation algorithm. This will be a good time to recollect from section 5.3.3.2 how a sender node gets feedback regarding whether a packet was dropped at the receiver or discarded due to errors using flags. In this figure, the light colored boxes indicate the pseudo code or the functionality of the code block.

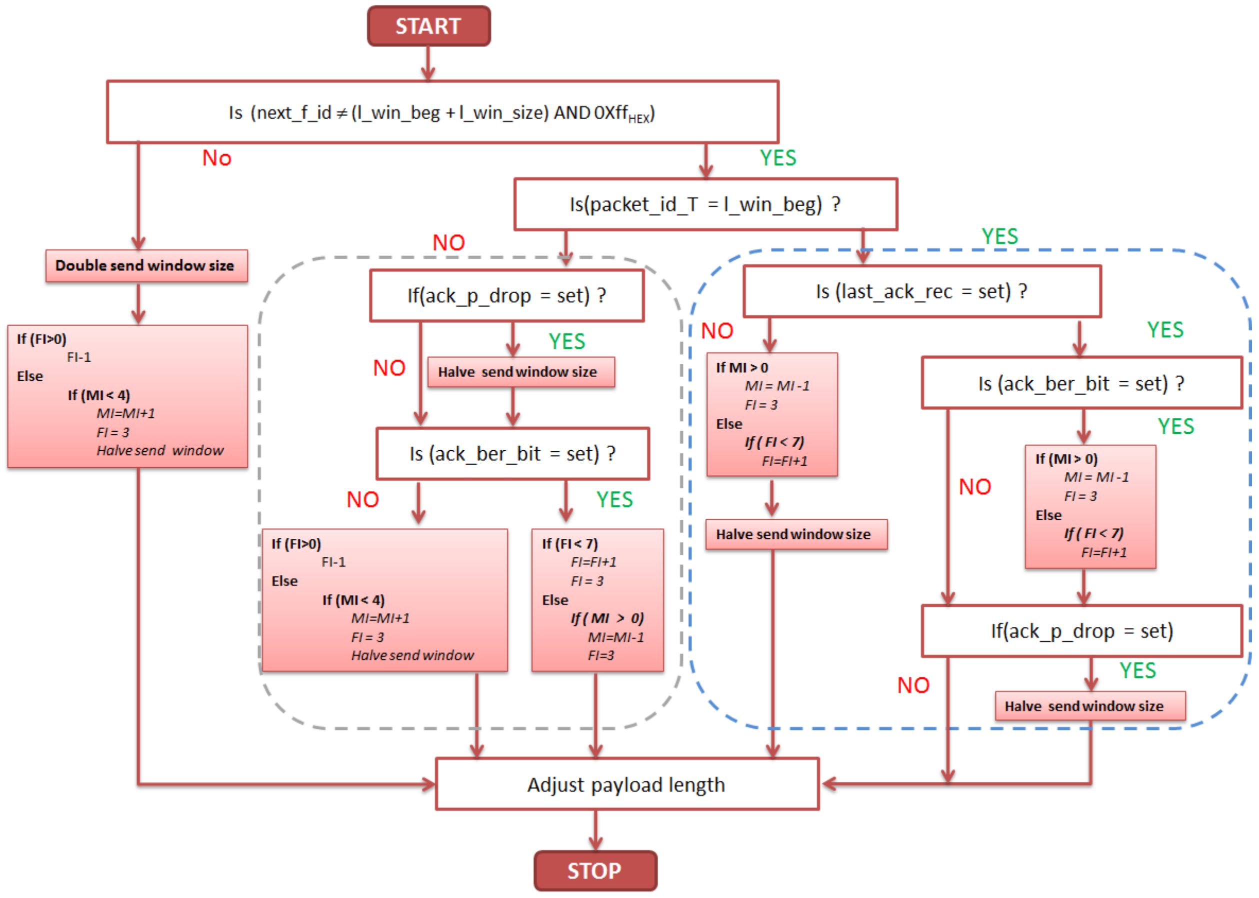

START
Is (next_f_id ≠ (l_win_beg + l_win_size) AND 0Xff_HEX)
No
YES
Is(packet_id_T = l_win_beg) ?
NO
YES
Double send window size
If (FI>0) FI-1 Else If (MI < 4) MI=MI+1 FI = 3 Halve send window
If(ack_p_drop = set) ?
YES
NO
Halve send window size
Is (ack_ber_bit = set) ?
NO
YES
If (FI>0) FI-1 Else If (MI < 4) MI=MI+1 FI = 3 Halve send window
If (FI < 7) FI=FI+1 FI = 3 Else If ( MI > 0) MI=MI-1 FI=3
Is (last_ack_rec = set) ?
NO
YES
If MI > 0 MI = MI -1 FI = 3 Else If ( FI < 7) FI=FI+1
Halve send window size
Is (ack_ber_bit = set) ?
YES
NO
If (MI > 0) MI = MI -1 FI = 3 Else If ( FI < 7) FI=FI+1
If(ack_p_drop = set)
NO
YES
Halve send window size
Adjust payload length
STOP

Figure 7.6: Rate adaptation Flowchart

If the most recently sent window of frames is received successfully by the receiving node, this algorithm takes the leftmost sequence of actions. If the same window of frames (i.e. two windows with the same first frame) is to be transmitted again, this algorithm follows the sequence of actions in the block denoted by dotted blue lines. If however only a partial number of frames from the most recently sent window need to be retransmitted, then this algorithm follows the sequence of actions in the block denoted by the grey colored dotted lines. It is to be noted that while adjusting parameters like modulation scheme, FEC scheme and send window size the values should not go below the defined minimum and maximum values.

Basically, the algorithm decides a course of actions based on whether a window transfer is fully successful, partially successful or fully unsuccessful. The pseudo code shows the actions in each such case. In case a window transfer is fully unsuccessful, the algorithm takes different actions based on the following scenarios:

- The first frame of the most recent window was erroneous.
- The first frame in the window was dropped.
- The acknowledgement was never received by the source node causing a timeout.

In case of i) the FEC scheme or modulation scheme is altered, in case of ii), the window size is halved, whereas, in case of iii), both of those actions take in effect. The rationale for the actions taken in case of iii) is that this scenario may occur because the request for acknowledgement was probably not even received by the destination node, or, the acknowledgement couldn't be recognized due to a bad channel.  Halving the window size reduces the chances of buffer overflow at the receiving node, reduces the number of retransmissions and increases frequency of an acknowledgement. Adjusting the FEC scheme and modulation scheme affects the robustness of data transfer to variations in channel.

Modulation is stepped down when a window transfer is fully unsuccessful or when the strongest FEC scheme at a higher order modulation scheme still yields errors causing partially successful window transfer. Modulation scheme is stepped up when a window transfer at a lower order modulation scheme using the weakest FEC scheme has been fully successful. The FEC scheme is stepped down (weaker FEC) when a window transfer is fully successful and stepped up (stronger FEC) when a window transfer is partially successful. Every time the modulation scheme is changed, a moderately strong FEC is chosen (index 3 from Table 7.4). Window size is halved when a frame is dropped at the receiver or when modulation is stepped up so that the number of retransmissions can be reduced if the higher modulation scheme yields errors. Window size is doubled when a window transfer is fully successful. The modulation scheme that the rate adaptation algorithm starts with is QPSK.

In order to ensure the defined maximum number of samples pushed during a in a frame, the length of the original data message is accordingly adjusted. Keeping the number of samples almost fixed (rather than variable) causes the frame length in time domain to be almost fixed. If the number of samples is fixed at 22000, the frame length will be 22000/1000000 = 22 ms.

As previously mentioned, this MAC protocol handles the continuity of the data. Changing the data message length while retransmitting frames can affect the continuity of the data read from a file (or data chunk). To visualize this problem, consider the scenario depicted in the following figure.

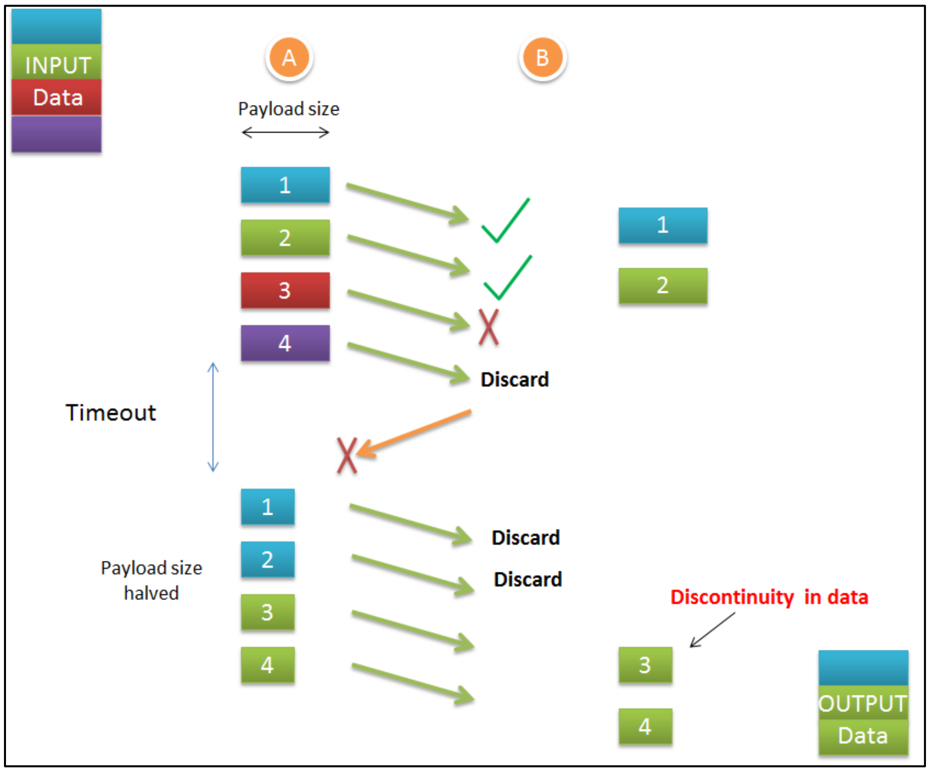


Figure 7.7: Rate adaptation and discontinuity in data

In the above figure, a node A sends a window of frames to node B. Node B receives the window partially and sends an acknowledgement for the frames. The acknowledgement however never makes it to node A because of severe channel conditions causing a time out at node A. Node A might fear that this could possibly be due to buffer overflow at the receiver, reduces the length of the data message in each frame and retransmits the window of frames. Node B receives the next expected frame in sequence which now has a different portion of the data. This causes data discontinuity. In order to avoid such a scenario, whenever a whole window is retransmitted due to a timeout, the payload length is not changed in the retransmitted window. Only the send window size can be reduced.

# Chapter 8

# Performance Analysis

## 8.1 Data transfer performance

Figures 8.1-8.4 reflect the practical performance of the MAC protocol. These graphs have been obtained by transferring frames with random data using this MAC protocol incorporating type1 HARQ at different channel conditions using different modulation schemes. For HARQ, two different types of convolutional codes have been used: 1) code rate=6/7, constraint length=7, Free distance=3 and 2) code rate=2/3, constraint length=7, free distance=6. No such FEC scheme is used for ARQ. Code scheme 2 is stronger than code scheme 1.

For HARQ, the acceptable packet error rate threshold is set to 7.5%. The size of one packet is 100 bytes or 800 bits while the frame length is fixed at 1600 bytes. This basically means that there can be at most one erroneous packet within the frame to avoid retransmission. The window size for all the trials is one. The modulation schemes used are BPSK, QPSK, 8PSK, 16QAM and a signal-to-noise ratio (or SNR) range of 4-13 dB is considered. This is because for the defined transmit power at each node, this is the achievable SNR range. As mentioned previously, the squelch threshold is defined at -30dB, 3dB above the noise floor. So, each trial has been carried out at received signal strength (RSS) of above -29dB.

The following performance metrics are defined to evaluate the MAC layer performance using HARQ.

- Goodput efficiency = $\frac{\text{Number of original message bits received successfully}}{\text{Total number of bits } (= \text{Redundant bits} + \text{original message}) \text{sent by the sender}}$

- Retransmission ratio = $\frac{\text{Number of frames retransmitted}}{\text{Total number of frames sent}}$

- Packet error rate = $\frac{\text{Number of erroneous packets}}{\text{Total number of packets received}}$

Each point on the graphs shown in Figures 8.1-8.4 has been obtained by carrying out 10 separate trials at around the same SNR. Each trial is continued till the number of successful frames received at the receiving node reaches 400. The maximum number of retransmissions for a frame is disabled. What this means is that the MAC protocol performs as many retransmissions as necessary to send 400 frames successfully. It is practically impossible to repeat a trial at exactly the same SNR. So, in order to obtain a point at a particular SNR, an SNR range of size .3 dB is considered. At the end of all the 10 trials for each point, the average value of the performance metric is computed and plotted at the average value of the SNRs over 10 trails. These graphs represent how the nature of the performance metrics are simultaneously affected as the channel conditions change. A good enough number of sample points are plotted to reflect the general trend in performance.

Each of the graphs in Figures 8.1-8.4 shows how the performance metrics for a given modulation scheme change with the FEC scheme at different SNR. We will observe the following trends in those graphs:

- As SNR increases, the PER decreases. Code 1 is weaker than code 2 as for the same SNR code 2 produces lesser number of packet errors. Not using any FEC scheme yields the worst performance of the three with higher packet errors at most SNRs.
- As SNR increases, the goodput efficiency increases. For QPSK, 8PSK and 16QAM, code 1 proves more efficient than code 2 at higher SNRs while code 2 is more efficient that code 1 at low SNRs. For BPSK, the weaker code 1 is always more efficient than the stronger code 1.
- As SNR increases, the retransmission ratio decreases. This is understandable because as the SNR increases, lesser number of packet errors occur. The retransmission ratio for the stronger code 2 is lower than that of code 1 as lesser number of packet errors occur using the former.

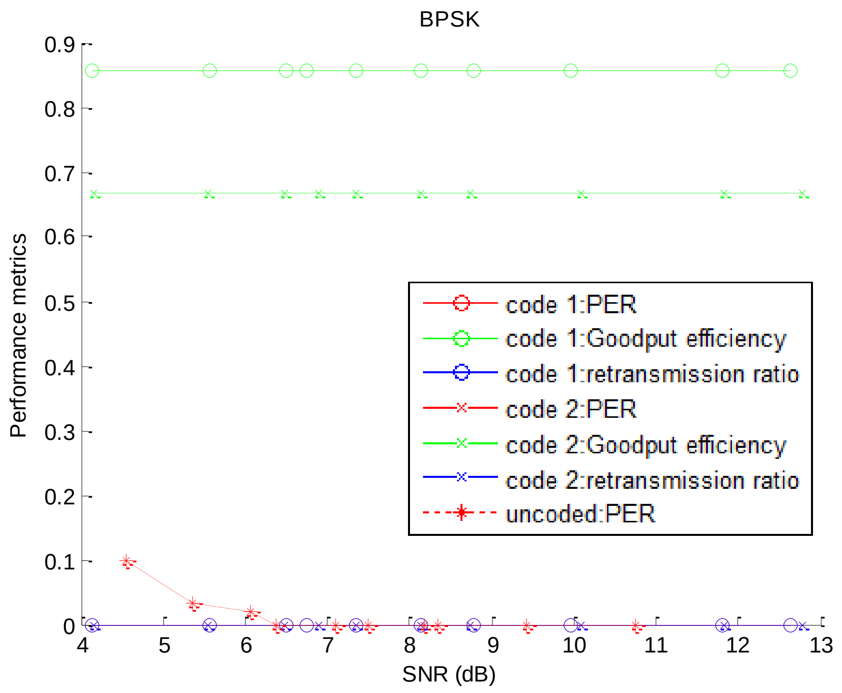


Figure 8.1: Influence on performance metrics (BPSK)

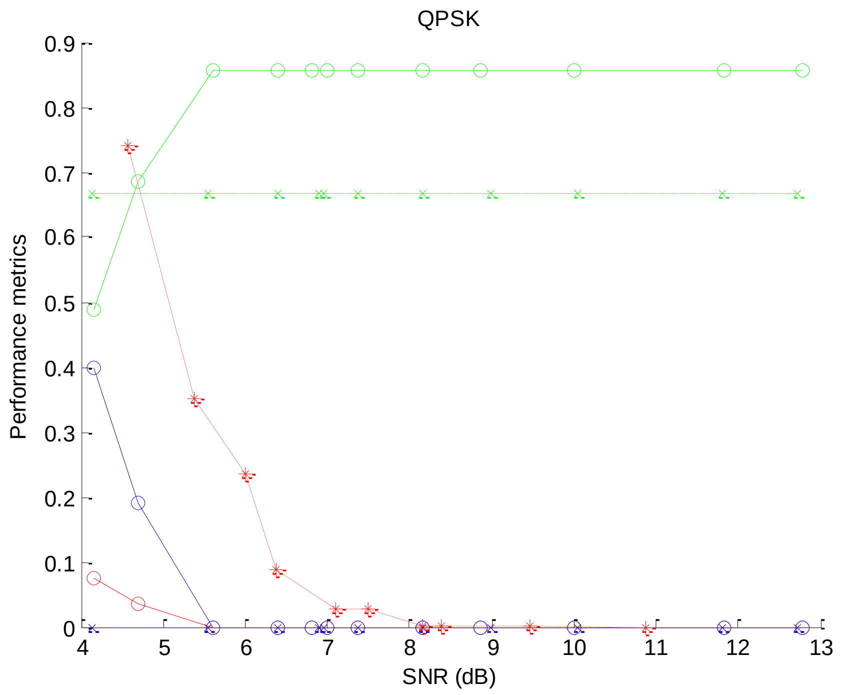


Figure 8.2: Influence on performance metrics (QPSK)

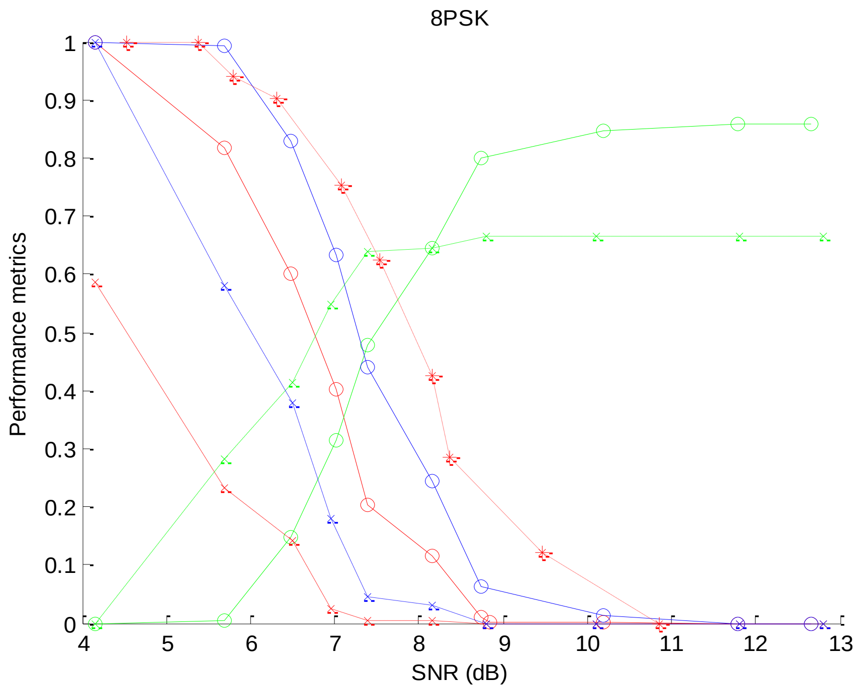


Figure 8.3: Influence on performance metrics (8-PSK)

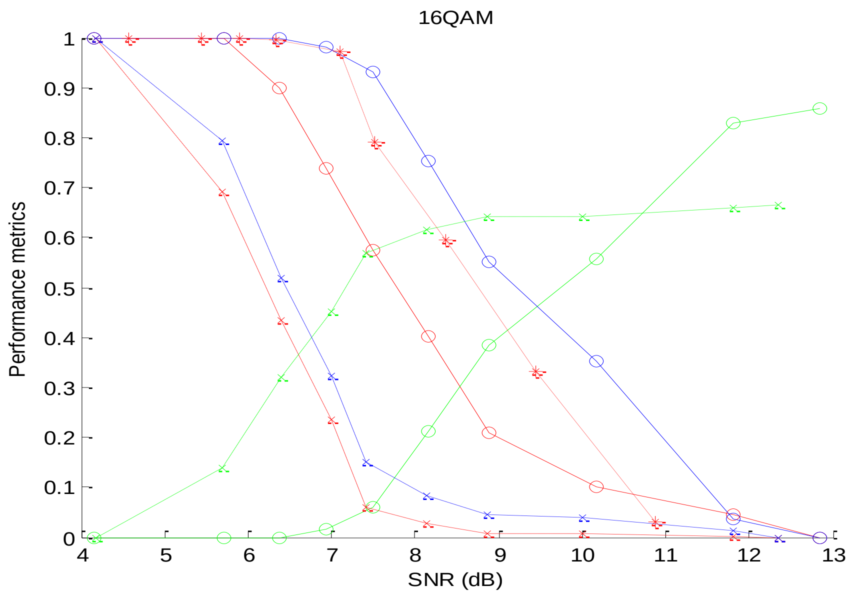


Figure 8.4: Influence on performance metrics (16-QAM)

The graphs in Figures 8.5-8.7 show how the nature of performance metrics changes with modulation scheme chosen. It is observed that the practical performance is on the lines of the expected theoretical results. The PER curves show that lower order modulation schemes are desirable at lower SNRs while higher order modulation schemes are desirable at higher SNRs. As the order of modulation increases, the goodput efficiency at lower SNRs decreases while at higher SNRs the goodput efficiency converges to a certain value depending upon the FEC scheme. Similarly, with increase in the order of modulation, the retransmission ratio increases. There are several factors that contribute in the evaluation of practical performance:

- Noise in the channel is not necessarily Gaussian in nature. Therefore synchronization at the receiver is not perfect.
- Because of randomness in channel conditions, it is not practically possible to achieve identical channel conditions for consecutive separate trails.
- Packet errors can also be induced by internal buffer overflows. For a multi-component software system, this cannot be ruled out.
- Hardware imperfections.

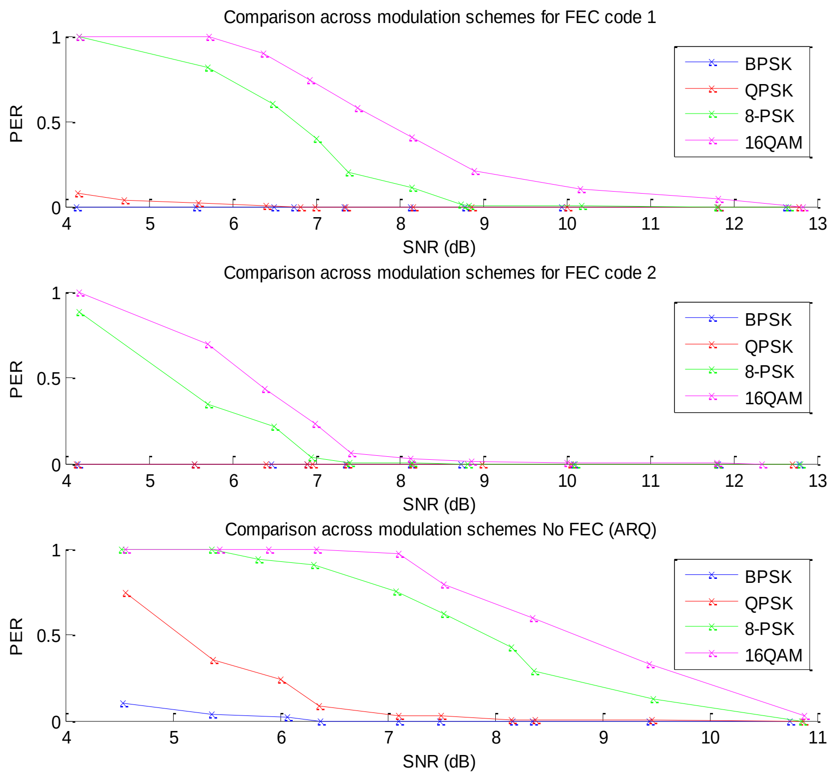


Figure 8.5: Effect of modulation on Packet Error Rate

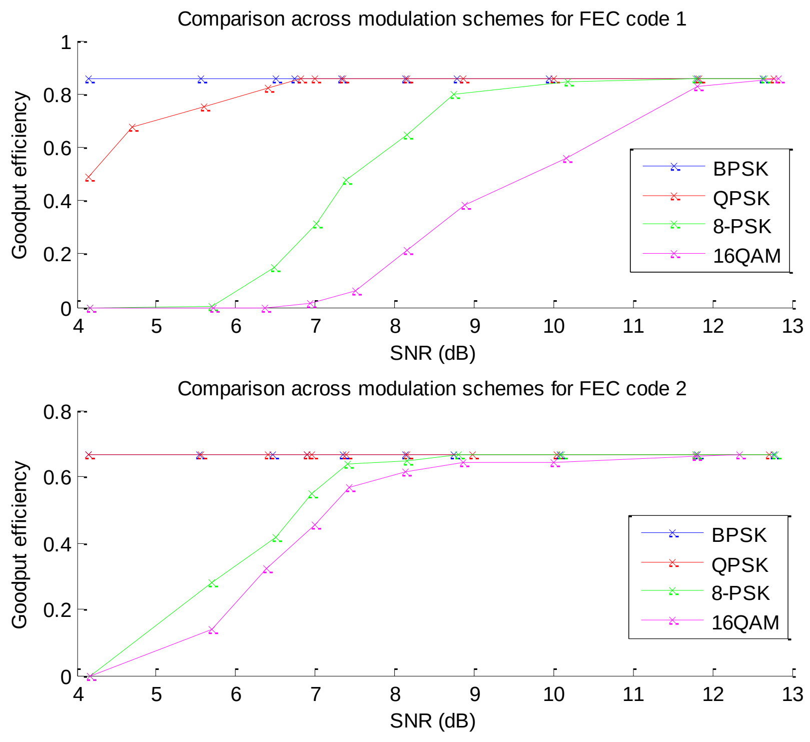


Figure 8.6: Effect of modulation on goodput efficiency

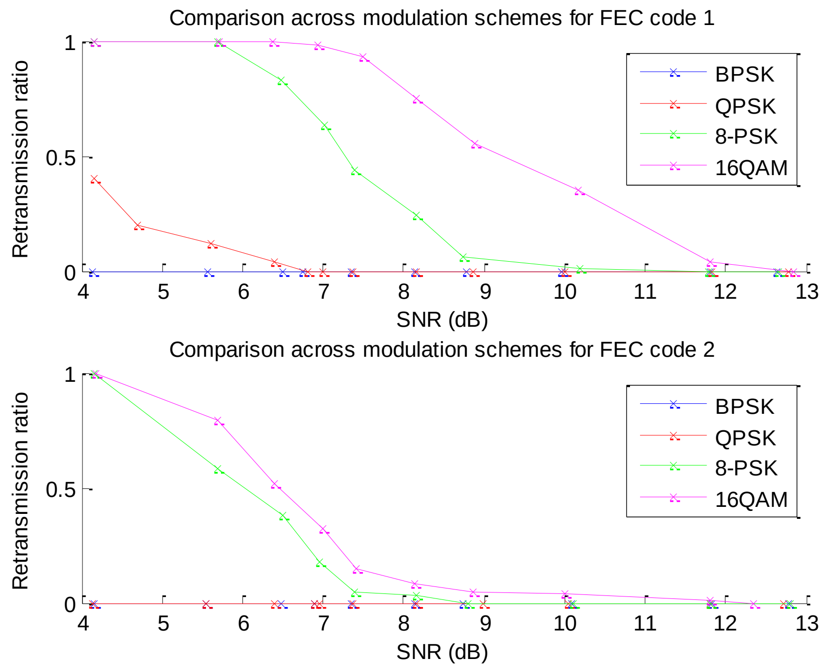


Figure 8.7: Effect of modulation on Retransmission ratio

## 8.2 Rate adaptation performance

A requirement for rate adaptation is feedback. The frequency of the feedback is constrained by the length of the send window which depends primarily upon the length of a frame. The faster it is, the faster rate adaptation will work. However, if the window size is small or the data message length is short, it affects throughput as well as efficiency as observed over practical experiments. While performing these experiments, two drastically different channel conditions were considered: Poor channel and Good channel. A channel is defined as poor when the received signal strength at the receiver is within 2dB above the squelch threshold (i.e. SNR ≤ 6dB) while SNR range of ≥ 10 dB is defined as good channel conditions. Figures 8.8-8.12 represent the performance of the rate adaptation mechanism in these defined channel conditions and contrast them against performance using a chosen fixed scheme in each type of channel condition. The graphs in these figures represent observations from four separate trials each pertaining to a combination of poor or good channel conditions and rate adaptation or fixed scheme. Each trial consists of sending a data file of size 125187 bytes from one node to the other. The average SNR obtained at the receiving node for poor channel conditions is 4.85 dB whereas the average SNR for the good channel conditions is 10.2 dB.

The fixed scheme is essentially same as Type I HARQ. In case of poor channel conditions the fixed scheme is: modulation scheme= BPSK; data message length = 800 bytes; FEC scheme: code rate =4/5, constraint length=9, free distance=3; send window=8; PER threshold = 7.5%. In case of the good channel, the fixed scheme used is: modulation scheme= 16QAM; data message length = 800 bytes; FEC scheme: code rate =4/5, constraint length=9, free distance=3; send window=8, PER threshold = 7.5%. It has been observed that the nature of the rate adaptation is similar over several trials in each of these channel conditions.

The performance metrics used to analyze the performance of the rate adaptation mechanism are defined as follows:

- Instantaneous transmission data rate = $\frac{Number\ of\ original\ message\ bits\ sent}{Transmission\ time\ of\ a\ frame}$

This is calculated at the beginning of the the first frame of every window. For a given window, the instantaneous transmission data rate is the same for all the frames in that window as each frame within a window uses the same modulation scheme and FEC scheme.

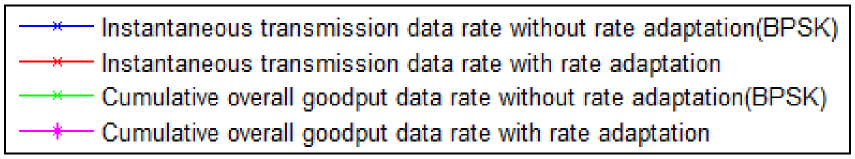
- Cumulative overall goodput data rate = $\frac{\textit{Total Number of original message bits received}}{\textit{Total time elapsed since start of data transfer}}$

  This is calculated after an acknowledgement is received at the end of evrey window.

- Actual bandwidth utilization = $\frac{\textit{Bandwidth used to send actual message bits in a frame}}{\textit{Total bandwidth used in sending a frame}}$

  This is calculated at the beginning of the first frame of every window.

Legend for Figure 8.8


Instantaneous transmission data rate without rate adaptation(BPSK)
Instantaneous transmission data rate with rate adaptation
Cumulative overall goodput data rate without rate adaptation(BPSK)
Cumulative overall goodput data rate with rate adaptation

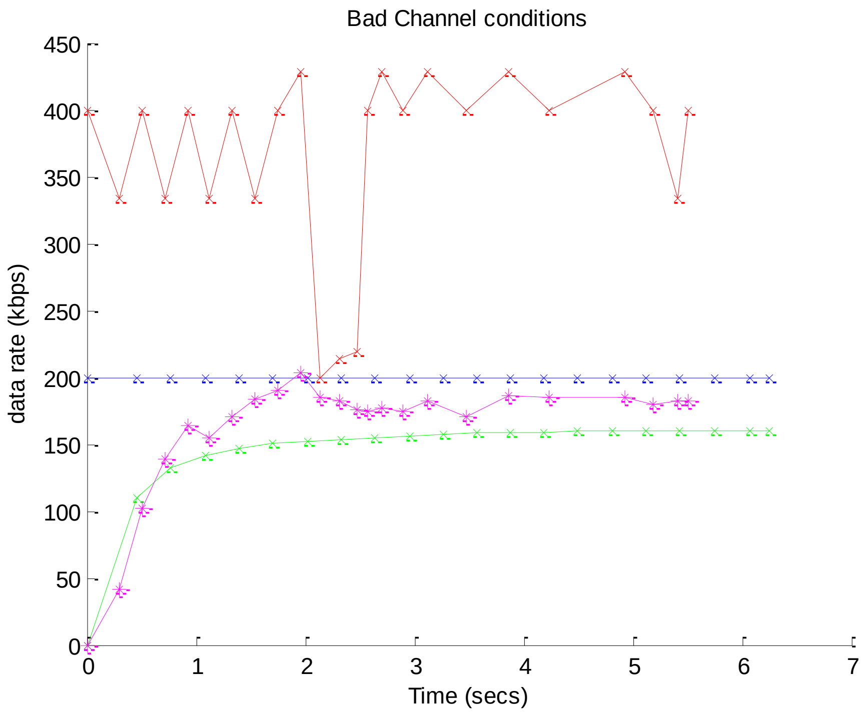


Figure 8.8: Rate adaptation under poor channel conditions

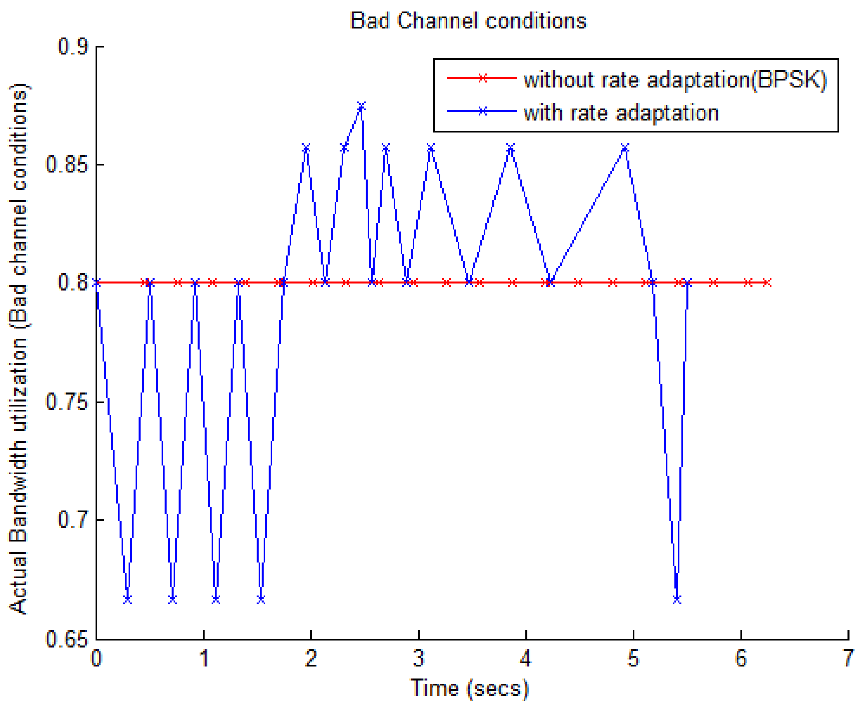


Figure 8.9: Bandwidth used in sending actual data under bad channel conditions

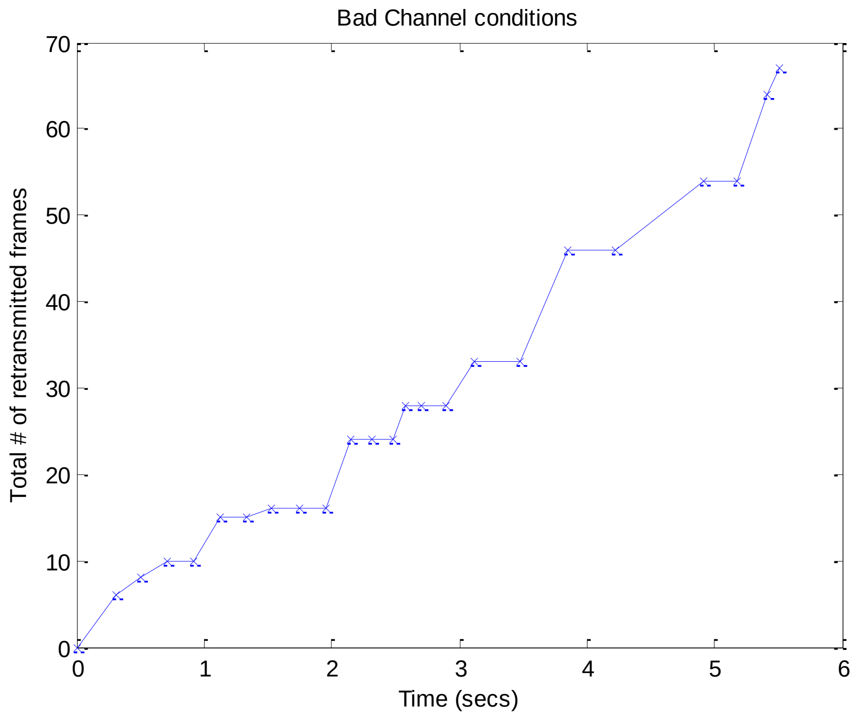
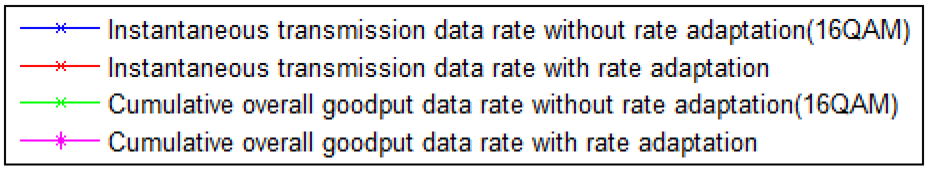


Figure 8.10: Number of retransmissions (poor channel conditions)

Figure 8.10 shows the cumulative number of retransmissions that take place while using rate adaptation. The number of retransmissions using the fixed scheme is zero and is not shown here. As rate adaption involved more retransmissions, it is less efficient than the fixed scheme. Figure 7.8 shows using rate adaptation mechanism, data transfer completes in a shorter duration of time thereby yielding a higher overall goodput data rate. The retransmissions however result in a lower efficiency in the performance of rate adaptation compared to the fixed scheme.

Legend for Figure 8.11



Instantaneous transmission data rate without rate adaptation(16QAM)
Instantaneous transmission data rate with rate adaptation
Cumulative overall goodput data rate without rate adaptation(16QAM)
Cumulative overall goodput data rate with rate adaptation

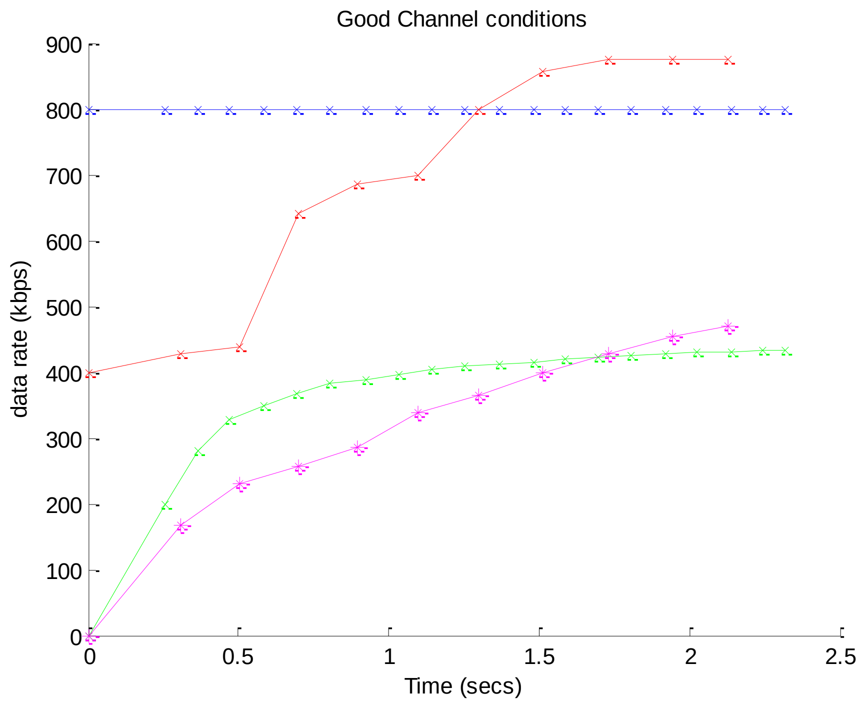


Figure 8.11: Rate adaptation under good channel conditions

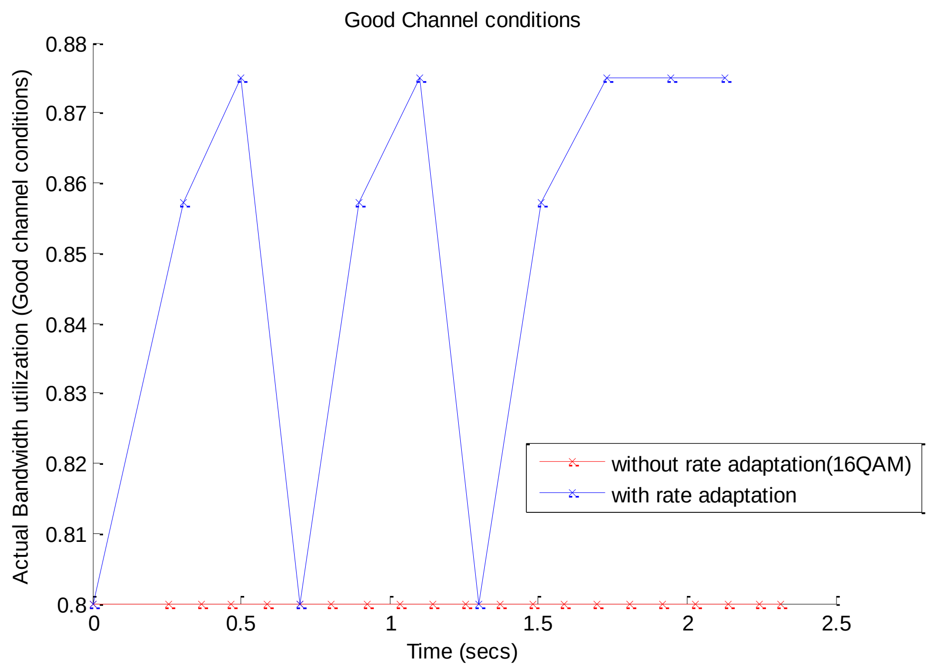


Figure 8.12: Bandwidth used in sending actual data under good channel conditions

Figure 8.11 shows that using rate adaptation, the data transfer completes in a shorter time thereby yielding a higher overall goodput data rate. In these channel conditions both the rate adaptation and the fixed scheme do not cause any retransmission. In Figure 8.12 one can see that the actual bandwidth utilization for rate adaptation changes in a pattern. This is attributed to the fact that the FEC scheme starts with index 3, switches to 2 after fully successful window transfer, then switches to 1 after fully successful window transfer and then the modulation scheme index is stepped up starting with FEC scheme index 3. This pattern continues till the highest modulation scheme with the weakest FEC scheme is chosen. It can also be noted that the distance between two successive sample points change with a pattern owing to the change in size of window.

An important thing to note especially in Figures 8.8 and 8.11 is that the overall goodput data rate obtained is almost half of the average transmission data rate both using and not using rate adaptation scheme. Another important question that arises from the above set of graphs is that if the fixed scheme maintains a fixed transmission data rate, then why the graph representing cumulative overall goodput data rate in Figure 8.11 is not a horizontal line. These observations lead to latency analysis during data transfer while using the MAC protocol.

## 8.3 Latency measurements

Latency measurements were performed on the trials involving fixed schemes in good and poor channel conditions.

- A timer is started just prior to sending the first frame of a window ($t_1$).
- The timer is switched off when an acknowledgement is received ($t_2$) after the complete window has been transmitted.
- No retransmissions take place in either channel condition.
- The transmission delay ($d_{trans}$) of each frame in the window is subtracted from ($t_2$-$t_1$).

$\therefore$ Round trip latency = ($t_2$-$t_1$)-(window size).$d_{trans}$

$$d_{trans} = \frac{\text{Number of samples in a frame pushed into USRP}}{\textit{Sampling rate}};$$

The value of $d_{trans}$ is also verified by measuring the length of a frame on the Spectrum analyzer in the timing mode. The latency is calculated at the end of each acknowledgement received and plotted over time to study any jitter. This is the round trip latency that includes processing delay of request for acknowledgement, processing delay of acknowledgement, processing delay of all the messages in the window at the receiver, delay due to traversal across OSSIE components, delay due to buffer handling at USRP, etc.

Figure 8.13 represents the round trip latency over time for two different modulation schemes. It also shows the effect of window size on this round trip latency.

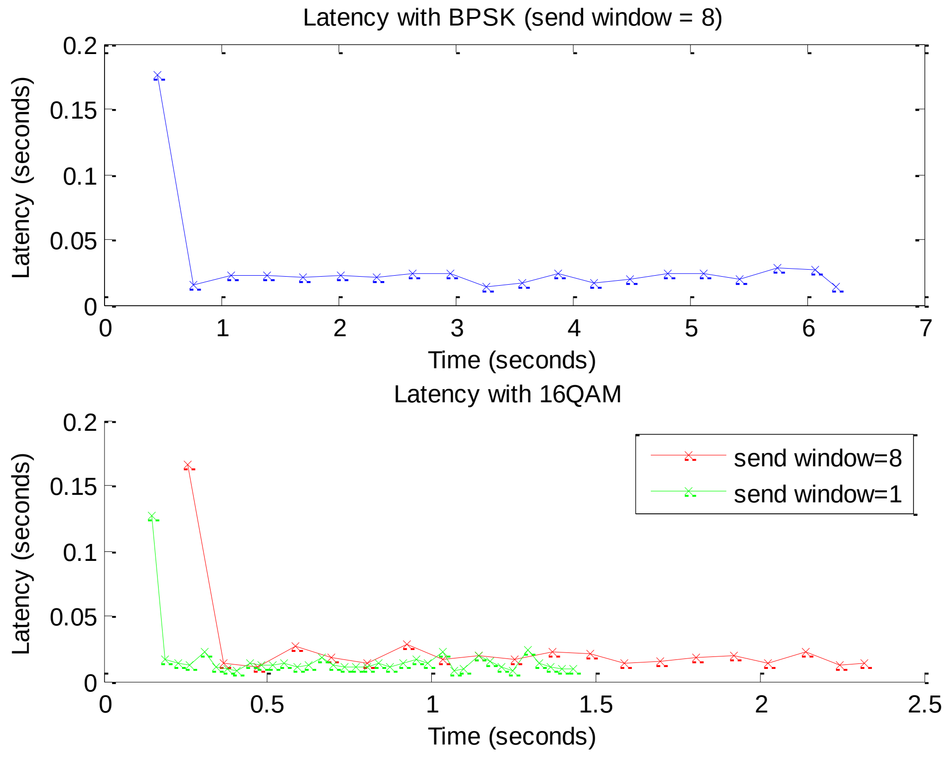


Figure 8.13: Round trip latency in window transfer

Note: A smaller data file is used in case of latency measurement for send window =1. As a result, the data transfer takes much shorter time to complete.

Statistics:

Mean latency using BPSK (window size = 8) = 20.9 ms

Variance in latency using BPSK (window size = 8) = 17.38 µs

Mean latency using 16QAM (window size = 8) = 17.8 ms

Variance in latency using 16QAM (window size = 8) = 23.7 µs

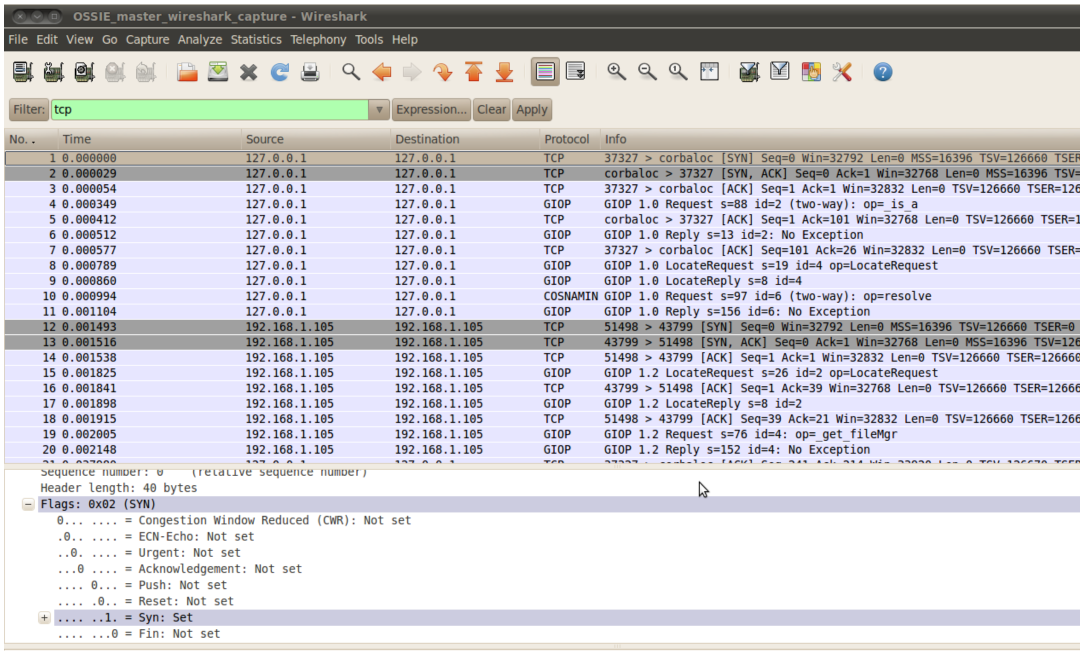
Mean latency using 16QAM (window size = 1) = 13.2 ms

Variance in latency using 16QAM (window size = 1) = 9.96 µs

It is to be noted that the mean latency for 16QAM decreases when the window size is reduced. This is because the processing latency decreases due to lesser number of frames in the window. There is not a significant change in the value of latency when the window size is changed from 8 to 1.This implies that the bulk of the latency is caused by other factor/s. BPSK has a higher average latency than 16QAM because for the same frame size, BPSK produces four times the number of samples as 16QAM which increases latency.

Another important revelation from the above graphs is that the first window always experiences the longest latency. This can be attributed to the underlying mechanism of how OSSIE components communicate with each other. The OSSIE components communicate with each other using TCP socket connection between internal ports (or sockets). CORBA is responsible for datatype handling (standard interfaces) and uses TCP to pass data between the components. As a result of this, every time the waveform is started, the TCP connection between every pair of ports goes through a slow start phase as part of the congestion control. More details on TCP congestion control can be found in [22]. The evidence for TCP socket connection establishment in the beginning of running the OSSIE waveforms is a screenshot of TCP packets captured using a network analyzer software called Wireshark [23] shown in Figure 8.14.

Experiments were performed to measure the time lag from the point when data is pushed out of a MAC component in the OSSIE waveform to the point when the data is actually transmitted by the USRP. The experimental setup is illustrated in Figure 8.15.


OSSIE_master_wireshark_capture - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: tcp
Expression... Clear Apply

```
No. . Time       Source         Destination    Protocol Info
    1 0.000000   127.0.0.1      127.0.0.1      TCP      37327 > corbaloc [SYN] Seq=0 Win=32792 Len=0 MSS=16396 TSV=126660 TSER
    2 0.000029   127.0.0.1      127.0.0.1      TCP      corbaloc > 37327 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 MSS=16396 TSV=
    3 0.000054   127.0.0.1      127.0.0.1      TCP      37327 > corbaloc [ACK] Seq=1 Ack=1 Win=32832 Len=0 TSV=126660 TSER=126
    4 0.000349   127.0.0.1      127.0.0.1      GIOP     GIOP 1.0 Request s=88 id=2 (two-way): op=_is_a
    5 0.000412   127.0.0.1      127.0.0.1      TCP      corbaloc > 37327 [ACK] Seq=1 Ack=101 Win=32768 Len=0 TSV=126660 TSER=1
    6 0.000512   127.0.0.1      127.0.0.1      GIOP     GIOP 1.0 Reply s=13 id=2: No Exception
    7 0.000577   127.0.0.1      127.0.0.1      TCP      37327 > corbaloc [ACK] Seq=101 Ack=26 Win=32832 Len=0 TSV=126660 TSER=
    8 0.000789   127.0.0.1      127.0.0.1      GIOP     GIOP 1.0 LocateRequest s=19 id=4 op=LocateRequest
    9 0.000860   127.0.0.1      127.0.0.1      GIOP     GIOP 1.0 LocateReply s=8 id=4
   10 0.000994   127.0.0.1      127.0.0.1      COSNAMIN GIOP 1.0 Request s=97 id=6 (two-way): op=resolve
   11 0.001104   127.0.0.1      127.0.0.1      GIOP     GIOP 1.0 Reply s=156 id=6: No Exception
   12 0.001493   192.168.1.105  192.168.1.105  TCP      51498 > 43799 [SYN] Seq=0 Win=32792 Len=0 MSS=16396 TSV=126660 TSER=0
   13 0.001516   192.168.1.105  192.168.1.105  TCP      43799 > 51498 [SYN, ACK] Seq=0 Ack=1 Win=32768 Len=0 MSS=16396 TSV=126
   14 0.001538   192.168.1.105  192.168.1.105  TCP      51498 > 43799 [ACK] Seq=1 Ack=1 Win=32832 Len=0 TSV=126660 TSER=126660
   15 0.001825   192.168.1.105  192.168.1.105  GIOP     GIOP 1.2 LocateRequest s=26 id=2 op=LocateRequest
   16 0.001841   192.168.1.105  192.168.1.105  TCP      43799 > 51498 [ACK] Seq=1 Ack=39 Win=32768 Len=0 TSV=126660 TSER=12666
   17 0.001898   192.168.1.105  192.168.1.105  GIOP     GIOP 1.2 LocateReply s=8 id=2
   18 0.001915   192.168.1.105  192.168.1.105  TCP      51498 > 43799 [ACK] Seq=39 Ack=21 Win=32832 Len=0 TSV=126660 TSER=1266
   19 0.002005   192.168.1.105  192.168.1.105  GIOP     GIOP 1.2 Request s=76 id=4: op=_get_fileMgr
   20 0.002148   192.168.1.105  192.168.1.105  GIOP     GIOP 1.2 Reply s=152 id=4: No Exception
```

```
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  − Flags: 0x02 (SYN)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...0 .... = Acknowledgement: Not set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
    + .... ..1. = Syn: Set
      .... ...0 = Fin: Not set
```

Figure 8.14: TCP packet capture while running an OSSIE waveform

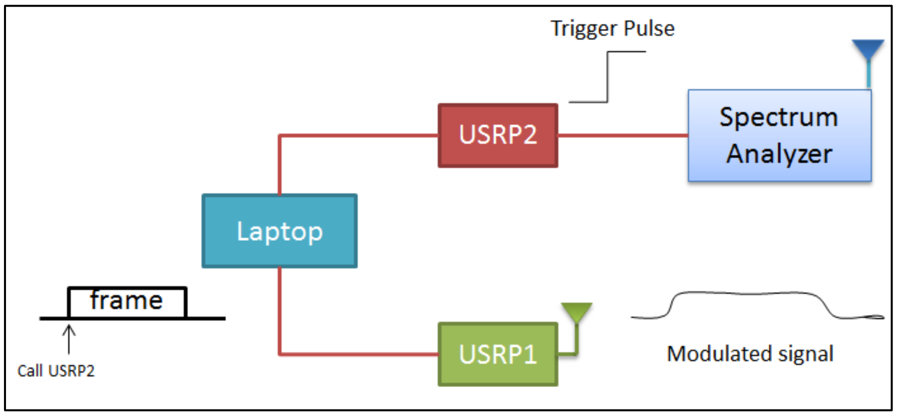
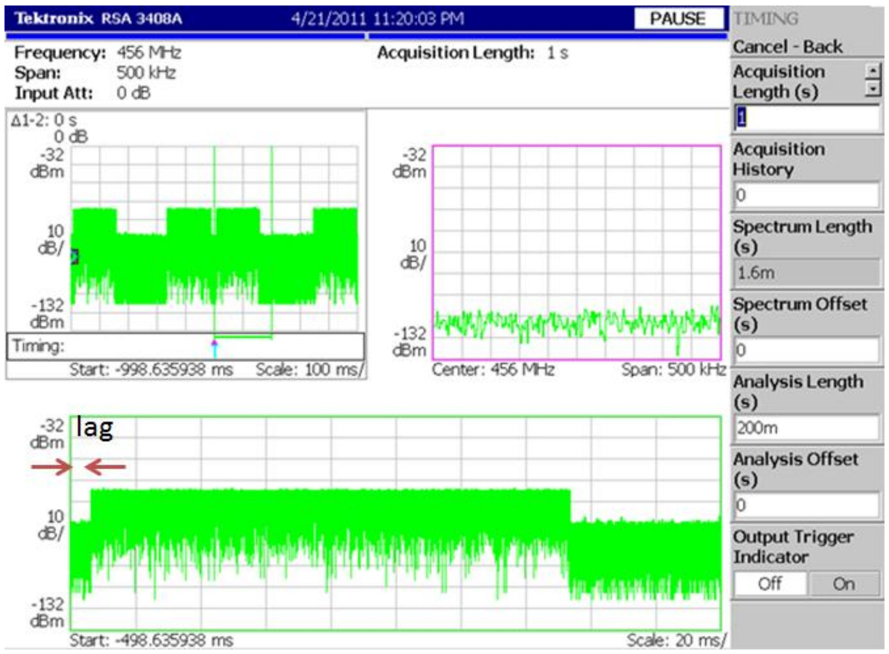


Figure 8.15: Latency measurement setup

As shown in the above figure, a function call is passed to USRP2 prior to sending a frame out of the MAC component in the OSSIE waveform. This function call basically prompts the USRP2 to send out an un-modulated pulse, which is fed into the spectrum analyzer. The spectrum analyzer is set to trigger to the pulse from USRP2. When the frame is actually transmitted by the USRP1, the spectrum analyzer detects it and displays a time domain signal with a timing reference to the pulse trigger. As a result, we can now determine the lag quite accurately. The following figure shows an annotated screenshot from the spectrum analyzer showing this lag.



Frequency: 456 MHz
Span: 500 kHz
Input Att: 0 dB
Acquisition Length: 1 s

Start: -998.635938 ms Scale: 100 ms/

Start: -498.635938 ms Scale: 20 ms/

Acquisition Length (s): 1
Acquisition History: 0
Spectrum Length (s): 1.6m
Spectrum Offset (s): 0
Analysis Length (s): 200m
Analysis Offset (s): 0

Figure 8.16: Transmitter side latency measurement

The average value of this one sided lag over 10 trials is ~5ms with a variance in the order of microseconds. This explains the round trip latency of over 10 ms for window size of one.

The following figure analyzes the major elements of delay in DL data transfer:

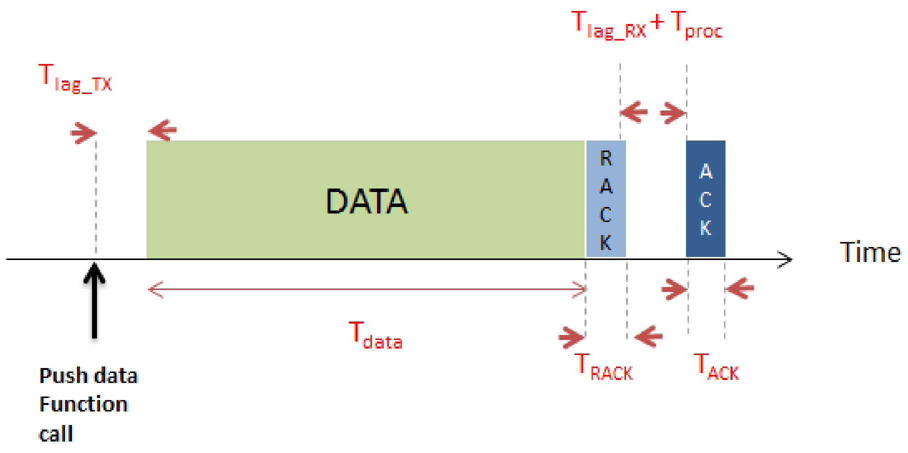


Figure 8.17: Latency components

From the above figure, $T_{lag\_TX}$ is the one sided lag on the transmitter side to send request for acknowledgement or RACK. $T_{lag\_RX}$ is the one sided lag at the receiving node in sending the ACK. $T_{data}$,$T_{RACK}$,$T_{ACK}$ are transmission delays due to the frame, RACK and ACK respectively. $T_{proc}$ is the processing delay at the receiving node. The latency mentioned previously includes $T_{lag\_TX}$,, $T_{lag\_RX}$ as well as $T_{proc}$..

## 8.4 Data rate analysis

$$\text{Throughput rate} = \frac{\textit{Number of bits successfully received}}{\textit{Total time taken}}$$

Figures 8.18-8.20 show how certain factors affect the throughput rates. These graphs are obtained by transferring a data file of size 128000 bytes using this MAC protocol. The channel conditions considered are good (SNR >10dB), so no retransmission takes place. Other parameters are kept fixed: modulation scheme=16QAM, send window size =8, FEC scheme index = 3.

### 8.4.1 Effect of data message length on throughput rate

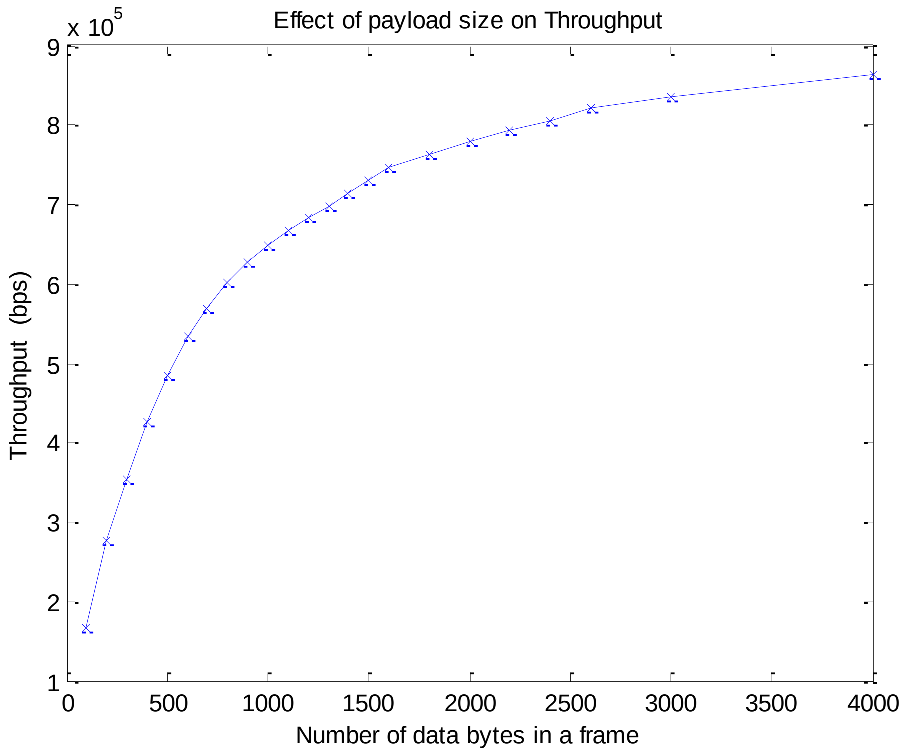


Figure 8.18: Effect of Payload size on Throughput

The above graph shows that as frame size is increased, the rate of increase of the throughput rate decreases. When the frame size exceeds above 5000 bytes i.e. 40000 samples (at 16QAM), the receiving side is not able to receive even a single correct frame and also experiences frame drops. This can be explained by the fact that when the number of samples pushed into the USRP exceeds 40000 samples, buffer overflow occurs at the FPGA in USRP. This is why the maximum number of samples that can be pushed into the USRP has set to safe 22000 in this MAC protocol. It is to be noted that sample point plotted on this graph is an average of 5 trials.

### 8.4.2 Effect of window size on throughput rate

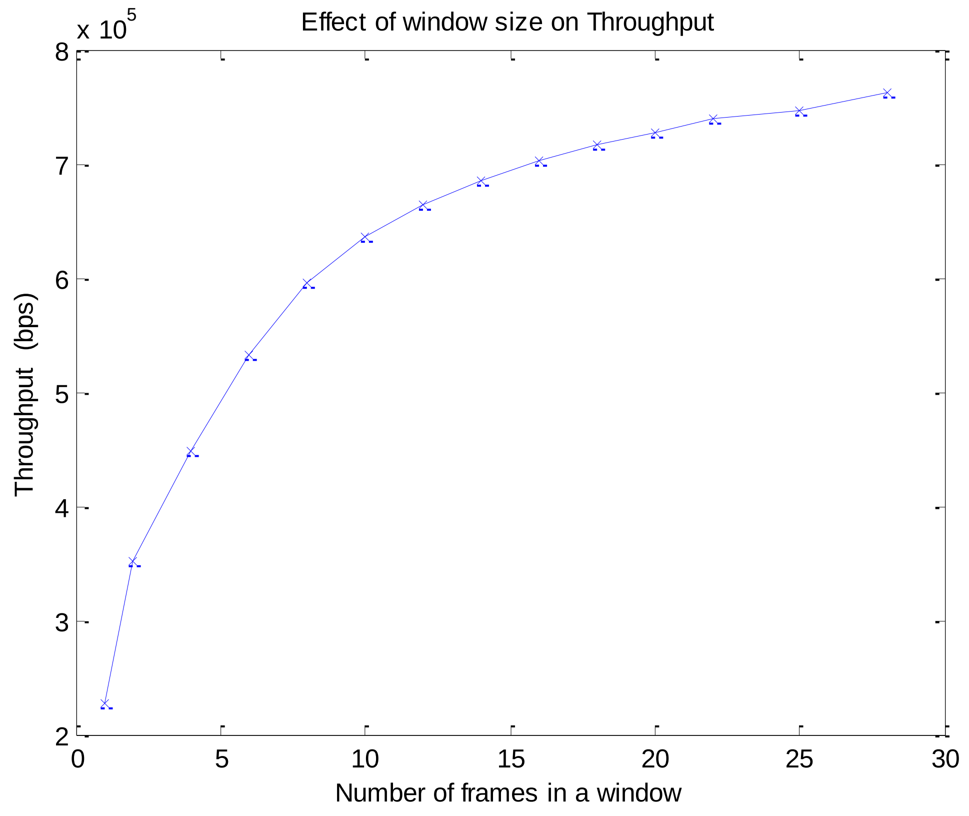


Figure 8.19: Effect of Window size on throughput

The above graph shows that as the window size is increased, the rate of increase of the throughput data rate decreases. In this MAC protocol, a maximum window size of 8 is considered to operate within the linear part of the graph. Also, every sample point plotted on this graph is an average of 5 trials.

## 8.4.3 Effect of void frame interval on throughput

As already mentioned in section 3.2.2.1, the *FlexframeSync* component sends a void frame to the subsequent MAC component in the receive chain every defined number of samples (void frame interval) for which the channel is detected as idle. The following graph shows the nature of relationship between the void frame interval and the throughput data rate achieved. Each sample point on the graph is an average of 5 trials. All other parameters were kept constant: modulation type = QPSK, Message size = 800 bytes, Window size = 8. The average SNR at which these trials were carried out is ~13dB.

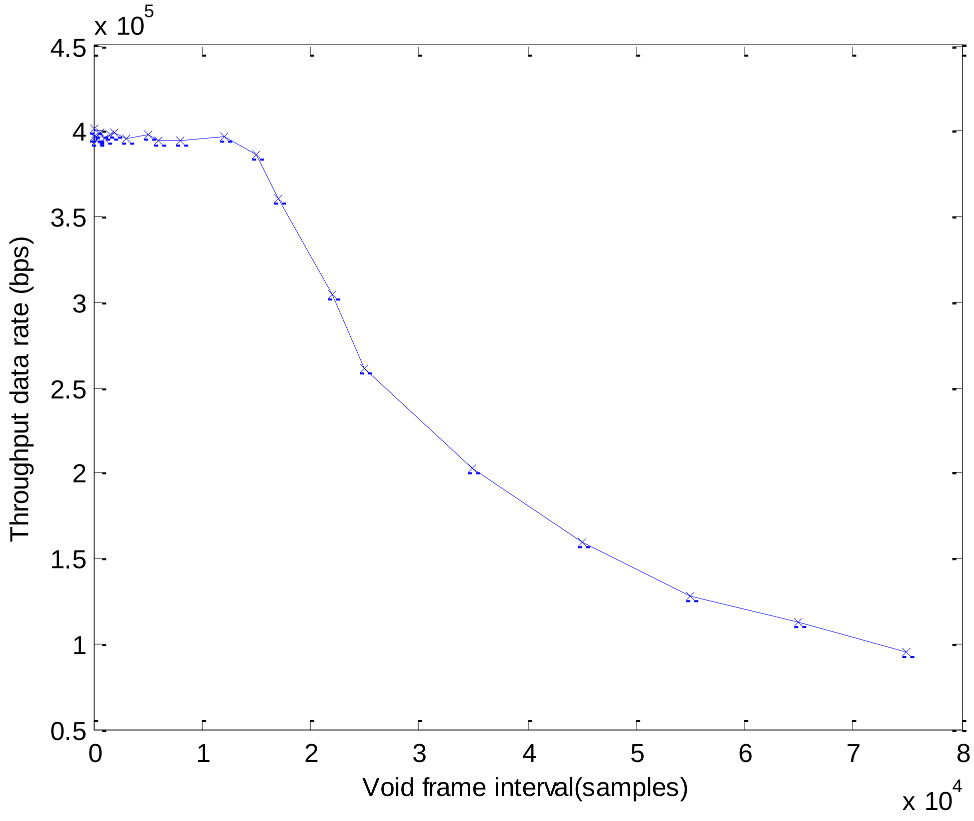


Figure 8.20: Effect of void frame interval on Throughput

## 8.4.4 Effect of zero-padding the USB blocks on throughput

As already mentioned in section 3.2.3, the native OSSIE USRP device driver is not suitable for packet data. The following table contrasts the difference in performance of the MAC protocol using the original OSSIE USRP device driver and the updated OSSIE USRP device driver.

Table 8.1: Original vs. New OSSIE USRP driver performance

| Case | | Original OSSIE USRP device driver | New OSSIE USRP device driver |
|---|---|---|---|
| 1) Without Rate adaptation Using QPSK Window size=8 | Number of RACKs | 39 | 20 |
| | Total time taken for data transfer | 9.174 seconds | 3.67 seconds |
| | Throughput data rate | 180.14 kbps | 360.24 kbps |
| 2) With Rate adaptation | Number of RACKs | 22 | 11 |
| | Total time taken for data transfer | 6.6715 seconds | 2.79 seconds |
| | Throughput data rate | 247.71 kbps | 592.34 kbps |

10 trials were considered under an average SNR of ~ 9dB. Squelch threshold is enabled (-30 dB, 3 dB above the noise floor). In all the above cases, total number of bytes transferred is 165264. It is clear that the new OSSIE USRP device driver enables more than twice the throughput data rate and is also more efficient in terms of number of control messages exchanged.

### 8.4.5 Comparison between the Uplink protocols

In order to compare the practical performance of the implementation of the two protocols discussed in Chapter 6, data transfer completion times were measured when two slave nodes (S1 and S2) sent data to the master node. The following table represents average values of 10 trails at average SNR of ~16dB. S1 sends a data file of size 80387 bytes while S2 sends a data file of 74999 bytes to the master node. Both the protocols were using the rate adaptation algorithm. It must be noted that in case of CSMA/CA the window size is always one, as a result of which the rate adaptation algorithm does not change the size of the window.

Table 8.2: Throughput comparison between Uplink protocols

| | CSMA/CA | Polling |
|---|---|---|
| Time of completion for S1 | 5.65 s | 2.54 s |
| Time of completion for S2 | 5.37 s | 2.72 s |
| Overall completion time | 6.17 s | 2.72 s |
| Overall goodput rate | 201.47 kbps | 456.85 kbps |

It can be observed that the Polling protocol gives much better data rate than the other. This is mainly because of the challenges faced in the implementation of CSMA/CA in a component based design on OSSIE. These pose a great challenge in mitigating collisions.

### 8.4.6 Effect of squelch threshold on throughput

For QPSK and BPSK, data transfer can take place without the squelch threshold being enabled. This is owing to that fact that these modulation schemes are quite robust to bit errors. As a result of 5 trails, the average throughput using QPSK with squelch threshold disabled is 352.54 kbps while the average throughput obtained with squelch threshold enabled is 397.06 kbps. Each trial pertains to data transfer of a data file of size 165254 bytes. The average SNR for all the trials is ~9 dB. Data message length in each frame is set as 800 bytes, window length is 8 and void frame interval is 5 samples. However, for the higher order modulation schemes like 8PSK and 16

QAM, the effect of squelch threshold is drastic. The data transfer doesn't even complete with the squelch threshold disabled because of incessant retransmissions on erroneous frames received.

## 8.5 Memory usage

Memory usage on a software radio affects total cost and power consumption. The memory usage of the OSSIE waveforms is analyzed using 'top' command in Linux operating system. The top command displays a summary of memory usage of individual process. Every component in the waveform is a separate process. Tables 8.3 and 8.4 show memory usage of the components during a trial of downlink data transfer.

Table 8.3: Memory usage at Slave node

| **Process** | **VIRT(kb)** | **RES(kb)** | **SHR(kb)** | **%CPU** | **%MEM** |
|---|---|---|---|---|---|
| FlexframeSync | 53896 | 6908 | 5188 | 21 | 0.2 |
| python | 87312 | 7676 | 3940 | 17 | 0.2 |
| USRP | 78052 | 8292 | 5292 | 8 | 0.2 |
| complexShort2Float | 53216 | 5956 | 4892 | 7 | 0.2 |
| slave_node_v3 | 75952 | .011 | 5116 | 7 | 0.3 |
| FlexframeGen | 54676 | 6556 | 5096 | 1 | 0.2 |
| complexFloat2Short | 53700 | 6452 | 4880 | 1 | 0.2 |

Table 8.4: Memory usage at Master node

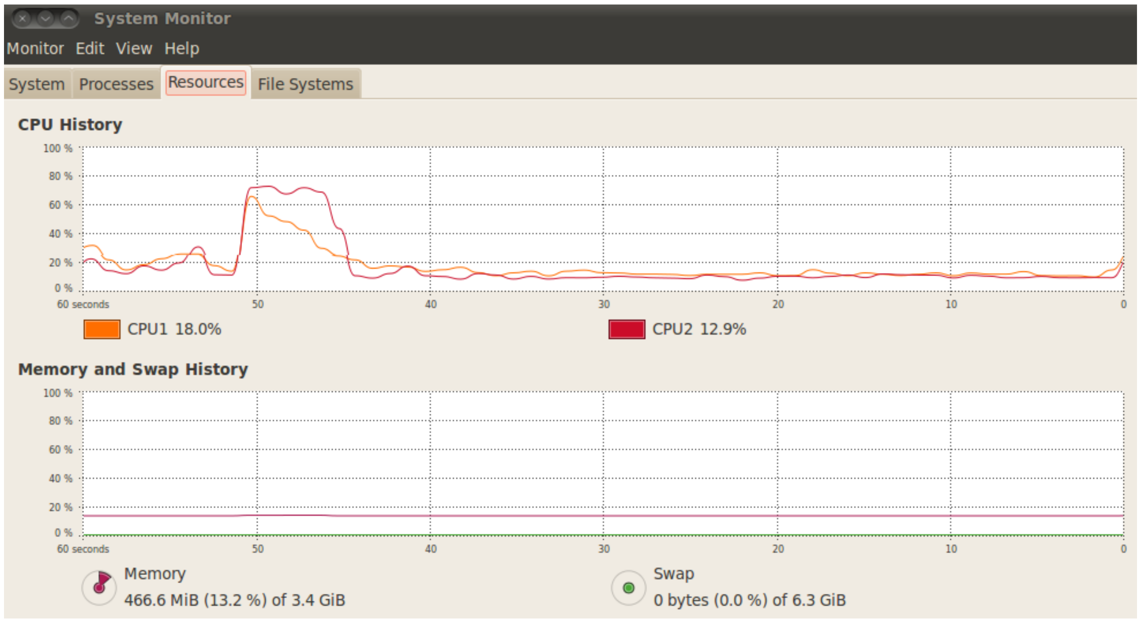
| **Process** | **VIRT(kb)** | **RES(kb)** | **SHR(kb)** | **%CPU** | **%MEM** |
|---|---|---|---|---|---|
| FlexframeSync | 54920 | 6668 | 5164 | 32 | 0.2 |
| python | 79012 | 7532 | 3940 | 17 | 0.2 |
| USRP | 78560 | 7724 | 5204 | 12 | 0.2 |
| complexShort2Float | 53348 | 6020 | 4892 | 9 | 0.2 |
| master_node_v2 | 84612 | 8284 | 7044 | 7 | 0.2 |
| FlexframeGen | 54404 | 6292 | 5100 | 2 | 0.2 |
| complexFloat2Short | 61700 | 6260 | 4992 | 0.0 | 0.0 |

Columns in the above tables:

- VIRT represents how much memory the process is able to access at the present moment. It refers to how much memory a particular application has requested but it is not necessarily the memory used by the process[24].
- RES stands for the resident size, which is an accurate representation of how much actual physical memory a process is consuming. (This also corresponds directly to the %MEM column.) This will virtually always be less than the VIRT size, since these processes depend on the C library[24].
- SHR indicates how much of the VIRT size is actually sharable (memory or libraries). In the case of libraries, it does not necessarily mean that the entire library is resident. For example, if a program only uses a few functions in a library, the whole library is mapped and will be counted in VIRT and SHR, but only the parts of the library file containing the functions being used will actually be loaded in and be counted under RES[24].
- %CPU represents a task's share of the elapsed CPU time since the last screen update, expressed as a percentage of total CPU time[24].
- %MEM represents a task's currently used share of available physical memory or the Random Access Memory (RAM) [24].

More tools to investigate memory usage can be found in [25]. A software tool called Gnome system monitor is used to graphically represent CPU usage at the master node during DL data transfer. The figure below is a screenshot obtained using this tool at the master node during a separate trial of the downlink transfer.

Figure 8.21 represents CPU and memory utilization in course of time. The laptop which is a part of the master node has two cores CPU1 and CPU2. While taking these readings, background application processes were killed. The memory usage graph is almost horizontal indicating that when the waveform is run, there is very little increase in the memory usage.



🟧 CPU1 18.0%
🟥 CPU2 12.9%

Memory
466.6 MiB (13.2 %) of 3.4 GiB

Swap
0 bytes (0.0 %) of 6.3 GiB

Figure 8.21: Resource usage graph

# Chapter 9

# Conclusions and Future work

## 9.1 Concluding remarks

The design of this MAC protocol has been employed in a practical setup prototyping a small scale wireless distributed computing network and every aspect of the design can be practically demonstrated. The MAC protocol uses a software infrastructure discussed in Chapter 3 which makes it modular and reconfigurable. In this work, all tests were conducted using an interpolation factor of 128 at a transmitting node and a corresponding decimation factor of 64 at the receiving node. If the value of the interpolation factor is lowered, the burden of processing at the USRP increases as it samples at a faster rate. The MAC protocol has been designed to operate at different compatible interpolation-decimation factor combinations. The optimizer block in the WDC architecture can harness this ability to achieve the desirable data rate or resource usage depending upon the application as well as radio hardware constraints.

The framing structure presented in Chapter 3 uses only eight bytes of MAC header which leaves most of the available bandwidth to be used to send data in the frame. A novel way of scheduling acknowledgements has been presented in the design of the Downlink Polling protocol in Chapter 4, which mitigates the latency constraints of the system as explored in Chapter 7. It is clear from Chapter 4 that scheduling the acknowledgements at the end of each window in a block of interspersed windows keeps the inter-frame-delay lowest in a round robin based polling scheme as well as provides higher efficiency and throughput rate of data transfer compared to using acknowledgements after every frame. In the polling protocols used in uplink and downlink, the use of window of frames rather than a single frame prior to requesting an acknowledgement is attributed to the round trip latency of at least 10ms.

Chapter 5 explores the implementation of polling protocol and CSMA/CA protocol on the uplink exposing the factors that affect the performance of the latter much more gravely as demonstrated in Chapter 7. It is understandably not feasible to implement efficient carrier sensing owing to such infrastructure and latency constraints. These constraints also support the use of a polling scheme over other channel access schemes in the MAC protocol for WDC.

The work presented in Chapters 4 and 5 shows that a discontinuity in data arises during retransmissions and can be solved easily by using temporary arrays to store and retrieve frame information from the most recent window. It is also shown in Chapter 6 that the rate adaptation algorithm used can cause a discontinuity in data as a result of a timeout for an acknowledgement and can be avoided in this scenario by not changing the data message length on retransmission.

Different aspects of performance of the MAC protocol are presented in Chapter 7, from which it is clear that the Packet Error Rate decreases with increase in SNR, increase in order of modulation scheme and decrease in code rate of FEC scheme. Goodput efficiency is higher for FEC with lower code rate at low SNRs while it is higher for FEC schemes with higher code rate at higher SNRs. At a particular SNR and FEC scheme, goodput efficiency decreases with the increase in the order of modulation. The retransmission ratio decreases with increase in SNR, decrease in code rate and decrease in order of modulation. Chapter 6 presents the working of a rate adaptation algorithm adopted to implement a variation of Type II HARQ. The practical performance of rate adaptation algorithm in Chapter 7 proves that while rate adaptation always provides higher throughput rate than Type I HARQ, Type I HARQ yields higher throughput efficiency at poor channel conditions (low SNRs : 4 -7 dB approximately).

Chapter 7 also brings to light the factors such as frame size, window size, void frame interval, transmit buffer handling and use of a squelch threshold that affect the data rate. These factors can be also be optimized by the optimizer block in WDC to govern resource allocation. The relationship between data rate and each of these factors is shown under good channel conditions (SNR > 10dB) to eliminate the effect of any retransmission. When the frame size is increased, the rate of increase of the throughput rate decreases or seems to converge to a certain value before plummeting to zero owing to buffer overflows at the USRP FPGA. As the window size is increased, the rate of increase of the throughput rate decreases. As the void frame interval increases the throughput rate remains fairly constant till around 10000 samples before falling. The zero padding of the USB blocks increases the throughput data rate by more than two times. This also reduces the number of control messages exchanged thereby reducing the bandwidth used in sending control messages. Enabling of the squelch threshold during data transfer yields a higher throughput for BPSK and QPSK while it enables a significantly higher throughput data rate for the higher order modulation schemes like 8PSK and 16 QAM.

This overall work offers transparency to the upper layer WDC architecture exposing the various parameters affecting data rate and latency which can be adjusted for efficient resource allocation. The MAC protocol makes the implementation of an upper layer WDC application feasible on a network of software defined radios. This MAC protocol can be used to test and develop future WDC applications on a network test-bed as well as compare and evaluate performances (ex-resources consumed) on different hardware or software platforms. This MAC protocol reinforces how software package like OSSIE can be effective in building functional applications for a network of devices.

## 9.2 Directions for future work

This work provides a benchmark MAC protocol which facilitates wireless distributed computing on a practical network of radio devices. As discussed throughout this document, there are several adjustable parameters in this MAC protocol: different types of timers and various factors that affect data rate. A prospective step to evolution of this protocol is optimization of this multitude of parameters to achieve a balance between resource consumption at a node and the data rate. A cross layer optimization between the MAC layer and the Physical layer can also be performed to yield better results. A cognitive engine can find use in the rate adaptation algorithm. The aspects of learning and decision making can improve the performance of the rate adaptation algorithm. In the future when open source libraries for Turbo codes become available, they can be used in place of convolutional codes to improve throughput data rate even in  poor channel conditions. This MAC protocol implemented as OSSIE waveform can be extended to low power embedded devices with ARM processor. Future work will also include integration of the MAC layer protocol with the upper layer WDC architecture.

# Bibliography

[1] (5/4/11). *IEEE Standards Association*. Available: http://standards.ieee.org/about/get/802/802.11.html

[2] (5/5/11). *Ettus Research LLC*. Available: http://www.ettus.com/products

[3] D. Datla, H. I. Volos, S. M. Hasan, J. H. Reed, and T. Bose, "Wireless distributed computing in cognitive radio networks," *Ad Hoc Networks,* vol. In Press, Corrected Proof.

[4] W. Ye and J. Heidemann, "Medium Access Control in Wireless Sensor Networks," in *Wireless Sensor Networks,* C. S. Raghavendra, K. M. Sivalingam, and T. Znati, Eds., ed: Springer US, 2004, pp. 73-91.

[5] F. A. Hamza. (2008, 5/2/11). The USRP under 1.5X Magnifying Lens! Available: http://gnuradio.org/redmine/attachments/129/USRP_Documentation.pdf

[6] D.Datla, S. M. Hasan, J. H. Reed, and T. Bose, "Fundamental Issues of Wireless Distributed Computing in SDR Networks," presented at the Wireless Innovation Forum Technical Conference, Washington DC, 2010.

[7] D.Datla, T.Tsou, X.Chen, S.Raghunandan, S. M. Hasan, and e. al., "Wireless Distributed Computing: A Survey of Research Challenges," *IEEE Communications Magazine,* under review.

[8] D. Datla, C. Xuetao, T. R. Newman, J. H. Reed, and T. Bose, "Power Efficiency in Wireless Network Distributed Computing," in *Vehicular Technology Conference Fall (VTC 2009-Fall), 2009 IEEE 70th,* 2009, pp. 1-5.

[9] D. Datla, H. Volos, S. M. Hasan, J. H. Reed, and T. Bose, "Task Allocation and Scheduling in Wireless Distributed Computing Networks," *Analog Integrated Circuits and Signal Processing (under review),* 2011.

[10] J.Reed, *Software Radio- A Modern approach to Radio Engineering*: Pearson, 2006.

[11] J. F.Kurose and K. W.Ross, *Computer Networking A Top-Down Approach*, Fifth ed.: Pearson, 2010.

[12] C. R. A. Gonzalez, C. B. Dietrich, and J. H. Reed, "Understanding the software communications architecture," *Comm. Mag.,* vol. 47, pp. 50-57, 2009.

[13] C. R. A. Gonzalez, C. B. Dietrich, S. Sayed, H. I. Volos, J. D. Gaeddert, P. M. Robert, J. H. Reed, and F. E. Kragh, "Open-source SCA-based core framework and rapid development tools enable software-defined radio education and research," *Comm. Mag.*, vol. 47, pp. 48-55, 2009.

[14] K. Saehwa, J. Masse, H. Seongsoo, and C. Naehyuck, "SCA-based component framework for software defined radio," in *Software Technologies for Future Embedded Systems, 2003. IEEE Workshop on*, 2003, pp. 3-6.

[15] (2011, 5/4/11). *GNU Radio*. Available: http://standards.ieee.org/about/get/802/802.11.html

[16] E. Paone, "Open-Source SCA Implementation-Embedded and Software Communication Architecture," Master of Science, School of Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm, Sweden, 2010.

[17] (2011, 5/4/11). *OSSIE SCA-Based Open Source Software Defined Radio*. Available: http://ossie.wireless.vt.edu/

[18] S. W. Smith. (2007, 5/10/11). *Digital Signal Processors*. Available: http://www.dspguide.com/ch28/4.htm

[19] P. Brenner. (1996, A Technical Tutorial on the IEEE 802.11 Protocol. *Breezecom Wireless Communications*.

[20] R. Comroe and D. Costello, Jr., "ARQ Schemes for Data Transmission in Mobile Radio Systems," *Selected Areas in Communications, IEEE Journal on,* vol. 2, pp. 472-481, 1984.

[21] (5/12/11). *DSP and FEC Library*. Available: http://www.ka9q.net/code/fec/

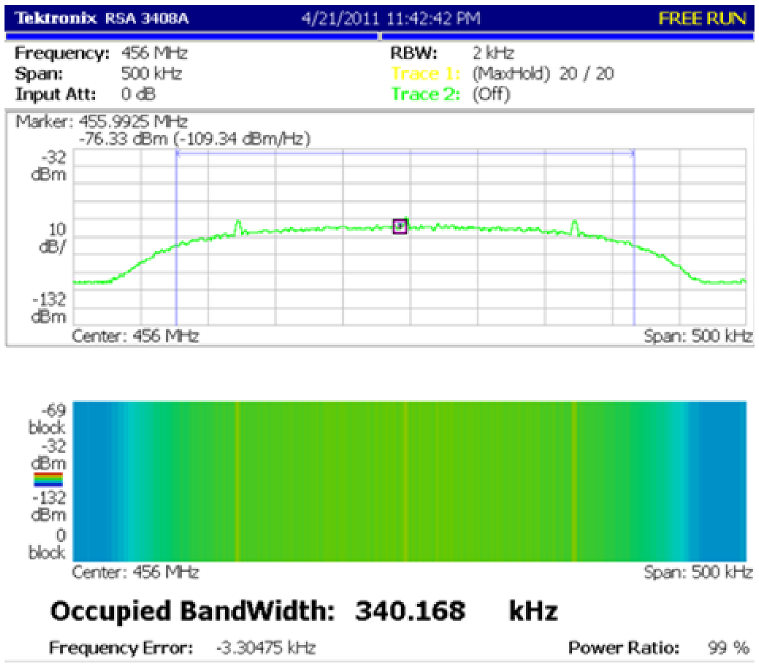
[22] (2001, 5/11/11). *TCP Congestion Control*. Available: http://www.ietf.org/rfc/rfc2581.txt

[23] (2011, 5/17/11). *Wireshark*. Available: http://www.wireshark.org/

[24] V. Gite. (2006, 5/16/11). How do I Find Out Linux CPU Utilization? Available: http://www.cyberciti.biz/tips/how-do-i-find-out-linux-cpu-utilization.html

[25] P. J. Balister, C. Dietrich, and J. H.Reed, "Memory Usage of a Software Communication Architecture Waveform," presented at the SDR Forum Technical Conference, Denver, CO, 2007.

# Appendix Figures


Tektronix RSA 3408A 4/21/2011 11:42:42 PM FREE RUN

Frequency: 456 MHz
Span: 500 kHz
Input Att: 0 dB
RBW: 2 kHz
Trace 1: (MaxHold) 20 / 20
Trace 2: (Off)

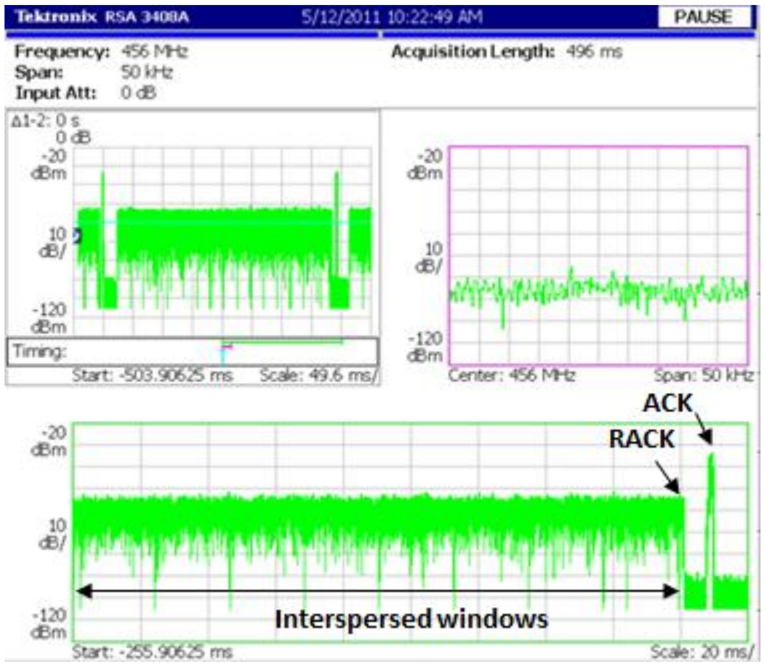
Marker: 455.9925 MHz
-76.33 dBm (-109.34 dBm/Hz)

Center: 456 MHz Span: 500 kHz

Occupied BandWidth: 340.168 kHz

Frequency Error: -3.30475 kHz Power Ratio: 99 %

Figure A.1: Occupied bandwidth during data transfer


Tektronix RSA 3408A 5/12/2011 10:22:49 AM PAUSE

Frequency: 456 MHz
Span: 50 kHz
Input Att: 0 dB
Acquisition Length: 496 ms

Figure A.2: Downlink data transfer (time domain)

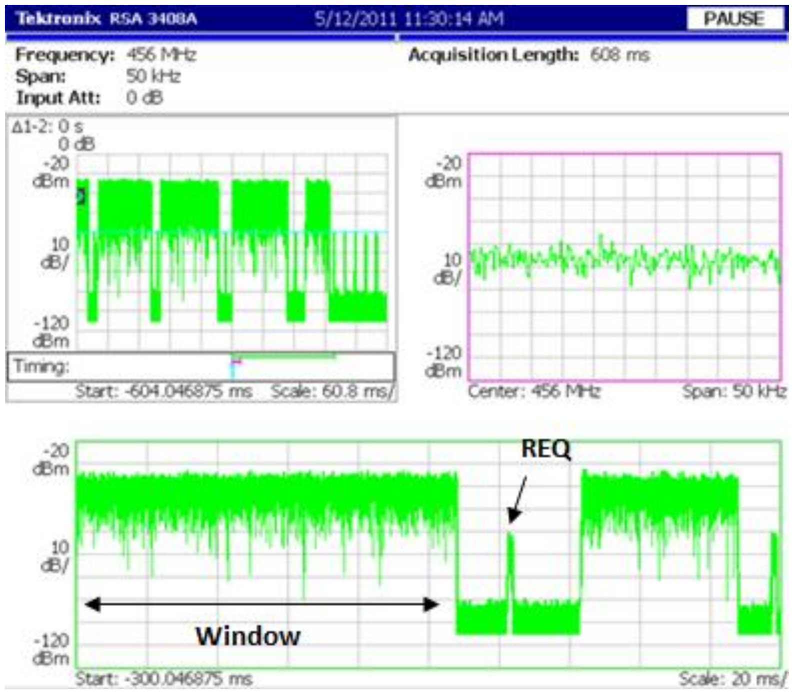


Figure A.3: Uplink data transfer (time domain) using polling protocol

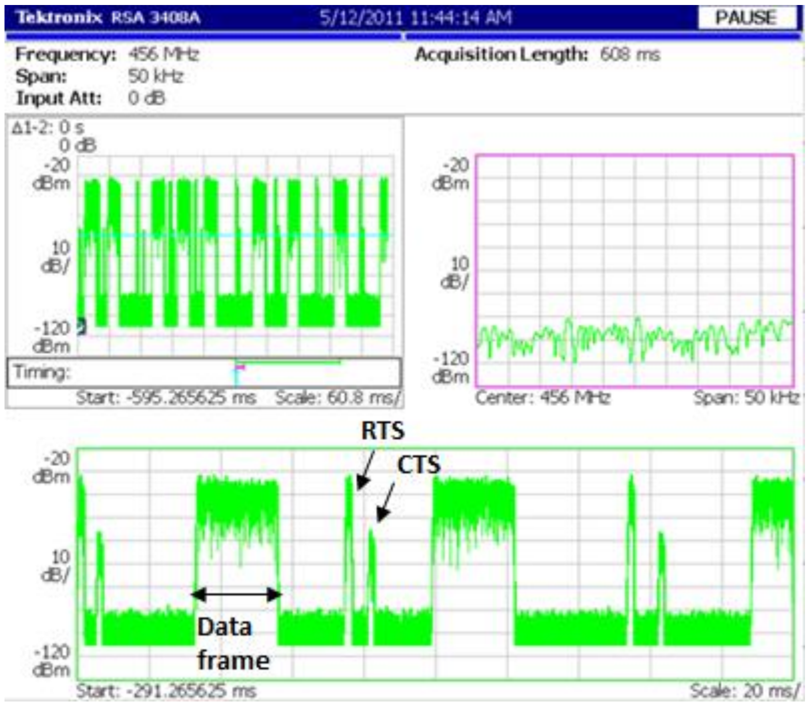


Figure A.4: Uplink data transfer (time domain) using CSMA/CA protocol