January 2011

NS-3 Simulation of WiMAX Networks

Christopher Thomas
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS
School of Engineering and Applied Science
Department of Computer Science and Engineering

Thesis Examination Committee:
Raj Jain
Chenyang Lu
Kilian Weinberger

NS-3 SIMULATION OF WIMAX NETWORKS

by

Christopher Thomas

A thesis presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

May 2011
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Simulation of WiMAX Networks and Allocation Systems

by

Christopher Thomas

Master of Science in Computer Science

Washington University in St. Louis, 2011

Research Advisor: Professor Raj Jain

Simulation is a powerful tool for analysis and improvement of networking technologies, and many simulation packages are available. One that is growing in popularity is NS-3, the successor to the popular NS-2. It is a significant departure from NS-2, and offers many advantages and disadvantages. In this thesis, we translate and update a sophisticated WiMAX simulation model from NS-2 to NS-3, and use this experience to investigate the major differences between NS-2 and NS-3, and the relative strengths of each package. We then use the NS-3 simulation model to provide analysis on a new WiMAX OFDMA downlink subframe mapping algorithm.

Acknowledgments

I would like to thank Professor Raj Jain for his constant help and guidance over the months I spent working on this thesis. I would also like to thank Professors Chenyang Lu and Kilian Weinberger for serving on my thesis committee.

Additionally, I would like to thank everyone involved in the development of the WiMAX Forum's NS-2 simulation model, which served as the basis for the NS-3 model that this thesis focuses on.

I would also like to take this opportunity to thank my friends and family, especially my parents for their continuous support and encouragement.

Chris Thomas

Washington University in St. Louis
May 2011

Contents

List of Tables

List of Figures

List of Abbreviations

BS	Base Station
BW-REQ	Bandwidth Request
CID	Connection IDentifier
CTS	Clear to Send
DL	Downlink
eOCSA	enhanced OCSA
FCH	Frame Control Header
FFT	Fast Fourier Transform
GMH	Generic MAC Header
IEEE	Institute of Electrical and Electronics Engineers
MAC	Media Access Control (Layer)
mOCSA	merging OCSA
OCSA	One Column Striping with non-increasing Area first mapping
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
PHY	PHYsical (Layer)
QoS	Quality of Service
RTG	Receive Transition Gap
RTS	Ready to Send
SS	Subscriber Station
TDD	Time Division Duplexing

TTG	Transmit Transition Gap
UL	Uplink
W-MAN	Wireless Metropolitan Area Network

Chapter 1

Introduction

There are many reasons that simulations are useful in the study and development of computer networks. For large-scale wireless networks, such as WiMAX networks, deployments are expensive and cover very large areas, making simulation models very important both for development and planning purposes.

The WiMAX Forum has developed a simulation model for use with the popular NS-2 simulator. While it is extremely popular, NS-2 has become somewhat dated, and a new simulator, NS-3, is being developed to replace it. Because there are significant architectural differences between the simulators, translating NS-2 models for use in NS-3 is an extremely involved process.

In this thesis, we discuss the process of translating the WiMAX Forum's NS-2 model to NS-3, and updating it to reflect a newer version of the WiMAX standard. This involves retrofitting the model to account for differences ranging from the programming languages used to define the model and specific simulation scripts to a vastly different packet architecture. The improvements involve changes to the transmission system to allow multiple nodes to send data simultaneously, as defined in newer revisions to the 802.16 standards WiMAX networks are based on.

The thesis is organized into seven chapters including this introduction. The second chapter provides a description of NS-2 and NS-3, highlighting differences between them. In the third chapter, an overview of WiMAX networks, with a particular focus on the elements involved in the changes to the simulation model is provided. The fourth chapter details the new NS-3 based WiMAX model. The fifth chapter provides details on the major issues encountered during the process of translating and improving

the model. The sixth chapter provides a sample application of the model to compare a new downlink-mapping algorithm to previously available alternatives. Finally, the seventh chapter summarizes the major points discussed in the thesis.

Chapter 2

NS-2 and NS-3 Systems

Both NS-2 and NS-3 are discrete event network simulators. This means that the simulation consists of a series of independent events that change the state of the simulation. Events are actions such as a packet being sent, a new node being added to the network, or a timer expiring. Each scheduled event runs until completion without advancing the simulation time, and then the simulation time is increased to the start time for the next scheduled event.

In both simulators, there is a core consisting of a scheduler and several useful classes defining nodes on a network, packets, and other similarly near-universal concepts. Various models use portions of this core package to implement specific network types, such as WiMAX, or simple wired Ethernet networks. Scripts then define network topologies of nodes connected using the networks defined in these models, and generate traffic between them.

However, while share this very basic architecture, they are very different. NS-2 is based heavily on the original NS, which started development in 1989 [6]. The architecture is starting to show it's age, and NS-3 was designed to avoid problems caused it [5]. Because the architecture changes are so significant, a decision was made to start from scratch rather than trying to update NS-2. This has resulted in a completely incompatible package that should be considered a successor to NS-2, rather than an evolution of it [4], and porting a NS-2 based model to NS-3 is an involved process. Significant differences include the programming languages used for development [8], the addition of a smart-object and memory management system [7], a more efficient and realistic data storage and packet system [7], a more realistic Node model [5], notable improvements to both memory usage and computational requirements to run a

simulation [10], support for industry standard trace files [4], and support for standard application interfaces such as POSIX sockets [5].

2.1 Programming Languages

NS-2 is implemented using a combination of oTCL, an object oriented extension to TCL, and C++. The core of the simulator and the various models are written entirely in C++, with the scripts describing the network topology and traffic generation written in oTCL by invoking objects automatically generated from the C++ base. This system was chosen in the early 1990s to avoid recompilation of C++ code, as that was very time consuming using the hardware available at that time [10]. Using oTCL for the much more frequently modified scripts allowed researchers to save time by very rarely needing to recompile the C++ base of the model.

Using oTCL does have significant disadvantages in that there is notable overhead introduced that causes scaling issues with large simulations [10]. As modern hardware makes compilation time less of an issue than when NS-2 was in it's early design phase, NS-3 can be developed entirely in C++. This also makes NS-3 somewhat more accessible to new users, as concerns about interfacing between multiple languages are eliminated, and only knowledge of C++ is required. A simulation script in NS-3 is written as a C++ program with a main() function, which is not possible in NS-2.

NS-3 does include limited support for Python in scripting and some related high-level tasks such as visualization. A set of bindings can be generated to allow a Python script to interact with the NS-3 API that would normally be accessible from a C++ script. If used, this reintroduces some issues that NS-3's removal of oTCL sought to avoid, but it is worth noting that using Python as a scripting language is optional in NS-3, and oTCL was the only available scripting language in NS-2.

2.2 Smart Objects and Memory Management

C++ Objects are reasonably simple compared to many newer languages. There is no automated garbage collection mechanism, and down casting is required to access members of subclasses. NS-2 requires basic manual C++ memory management. Because NS-3 is implemented in C++, all normal C++ memory management functions such as new, delete, malloc, and free are still available. However, the NS-3 core module includes several classes that can be used to automate these processes [7].

The ns3::Object class serves as the parent (or in many cases grandparent or higher) of most classes in NS-3 models. It contains functions to allow for reference counting with automatic deallocation of the object when the reference count reaches zero. This is especially useful when dealing with Packet objects, which are frequently created, destroyed, and copied when processing traffic.

NS-3's Object class also provides an aggregation system, by which Objects can be attached to other Objects at runtime [5]. This is useful for removing bloat from classes like Node. The NS-3 node has, by default, very little included. Other objects such as NetDevices (interfaces), internet stacks, and routing protocols are added only as needed. This means that, for example, a node on a wired network that has no use for location information does not waste storage space with parameters to track that. Similarly, if a user requires a customized internet stack, that user simply aggregates the custom class to the Node instead of the default IPv4-based stack, and there is no ambiguity or confusion over which is present.

2.3 Packets

In NS-2, a packet consists of two distinct regions. The first is reserved space for headers, and is shown in Figure 2.1; the second stores payload data. This includes a header common to all NS-2 packets including such data as a parameter specifying the

amount of data considered to be stored in the packet object (including headers) that has no direct relation to how much data is actually used. By default, the header region includes all headers defined as part of the protocol in use, regardless of whether or not that particular packet will use that particular header. This is done in part because NS-2 never frees memory used to store packets until the simulation terminates, but instead reuses the allocated packets repeatedly. Therefore, there must be space for not only the current header, but any header that may potentially be needed when that packet allocation is reused any number of times during the remainder of the simulation.

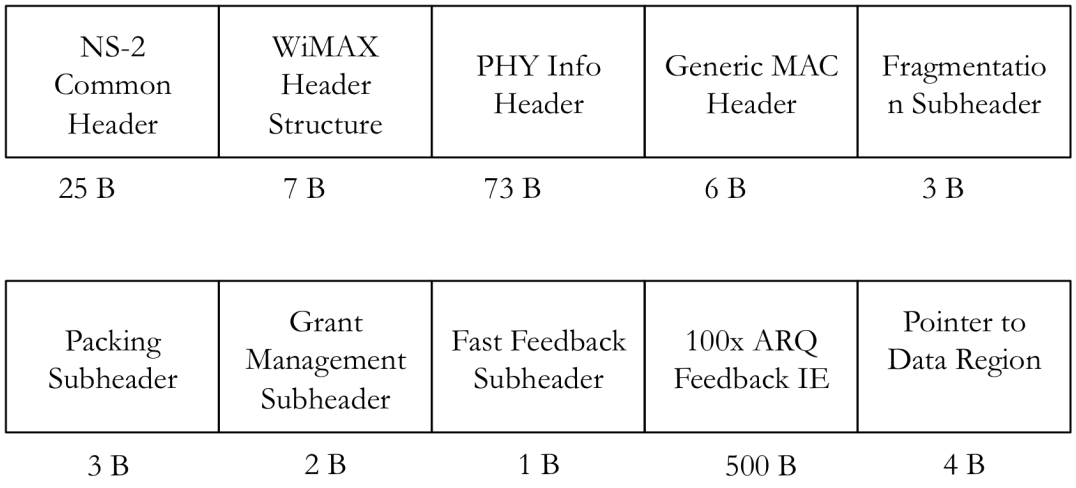

Figure 2.1 NS-2 Packet Structure

Because it is trivial to access the location a specific header will be stored if present, it is trivial to determine exactly which headers are attached to a given packet. In an actual network where what comes off the connection is basically just a stream of bits, the packet must be decoded and the headers that are present at the start allow a node to determine which headers are and are not present.

The data region of an NS-2 packet is dynamically allocated, with a void pointer provided to access the new data. By casting this pointer to whatever structure the developer wishes to add to the packet and then setting the members of the structure to

the desired values, access to this region is very simple. This system is very similar to using malloc to dynamically allocate memory from the heap. Directly accessing the data in this way also allows the values to be easily and quickly changed.

In many cases the structures allocated into the data region of a packet cover a variety of information which may or may not included in that particular instance of the packet type. For example, in the NS-2 WiMAX simulation, there is a structure representing a single bandwidth grant in an allocation map. In most cases this information element is only 4 bytes in size, but in some cases, as defined based on the value of one of the members of the structure, more data may be included. The structure contains fields for every possible point of data, for a total of 184 bytes. In the NS-2 model of a packet this does not especially matter, as the field defining the total packet size as far as the simulation cares will only be increased by 4 bytes unless more data is required, and that is what the scheduling and transmission logic will work with.

At no point is the specific stream of bits that would be transmitted over a real network determined. Instead, the data is added in whatever order and with however much extra as is convenient, and the size that the stream would be is maintained.

The NS-3 approach to packet storage is extremely different from NS-2's. A packet consists of a single buffer of bytes, and optionally a collection of small tags containing meta-data. This means that when a packet is received, or even when it is passed around internally in a given node, there is no easy to way to determine what headers or data is or is not present, or where any present data is located in that buffer. The idea is that the buffer corresponds exactly to the stream of bits that would be sent over a real network. All information that is to be added to the packet is done by use of subclasses of either Header, which adds information to the beginning of the buffer, or Trailer, which adds to the end. These classes consist of whatever data storage is convenient when working with them, and several specific functions to write the data to or read it from the byte buffer. More information on exactly how this works can be found in Section Chapter 4.

Unlike NS-2, there is generally easy way to determine if a specific header is attached. Knowledge of both the headers that have been serialized into the buffer and the order in which they were added is necessary to access a given header. To modify the contents of a specific header, a developer must remove all headers added after it, then remove it from the packet, modify it as desired, and add everything back in the reverse order.

The size that the simulator considers the packet to be is determined by the size of the buffer. Since adding a header or trailer adds exactly and only the bytes that would really be transmitted, there is no need to maintain a count of the size outside of this. This system also results in significantly less wasted memory, as each packet does not contain empty space that could be used for every possible header, or for portions of included frame structures that are unused. In the example given in the previous subsection regarding the 184-byte structure when only 4 bytes are actually used, in NS-3 only the four bytes are generally added. Only in the rare cases where one of the expanded forms of the information element is needed will that extra data be added.

2.4 Nodes

To facilitate easily creating realistic network topologies, NS-3 uses a Node system designed to emulate real computers [5]. A Node in NS-3 is very basic at first, and then using the aggregation system described in Section 1.2, NetDevices, Applications, Stacks, and other objects are added in much the same way that components would be added to a computer in a real network. A NetDevice can be viewed as network interface hardware, Applications that run on nodes interface with Stacks (which, in turn, interface with NetDevices) using an API that closely resembles the implementation of sockets on Unix systems [7]. Much like on a real system, different NetDevices will work with different stacks depending on whether they expect IPv4, IPv6, or any other stack-dependent parameters.

One of the design goals of this system was to make NS-3 code portable with real devices [11], something which was not easy with NS-2. NS-2's Node model needed to be specifically subclassed to add much of this functionality, and the interfaces were different enough that code reuse between real applications and simulation was not common.

2.5 Performance

NS-3 offers substantially and consistently superior performance both in required computation and memory footprint compared to NS-2 [10]. The source of the memory footprint gains are fairly straightforward, as discussed above the aggregation system prevents unneeded and sometimes very large parameters from being stored when they are unnecessary, and packets do not contain large amounts of meta-data and unused, reserved header space.

The total computation time required to run a simulation scales better in NS-3 than NS-2. This has been attributed to the removal of the overhead associated with interfacing oTcl, and the overhead associated with the oTcl interpreter.

2.6 Simulation Output

One of them most useful tools available for presenting the results of a simulation is animation. Generally speaking, an animation package would be able to show both the network topology and data flow through that topology. This can either be displayed as the simulator is running or after the fact from a trace file generated by the simulator.

NS-2 comes with a package called NAM, or The Network Animator. It is a Tcl based animation system that processes a specialized trace file generated by NS-2 and produces a visual representation of the network described. As it is based on a trace file, the animator can easily control the exact speed of simulation, from slow enough to see

individual packets to fast enough to see overall throughput. It is also very simple to rewind the animation, as it can simply move backwards through the file.

NS-3 employs a package known as PyViz, which is a python based real-time visualization package. It takes input directly from trace hooks in the simulation as it runs rather than parsing a trace file after the fact. This comes with several advantages and disadvantages. The first notable advantage involves the fact that, as the simulation data is displayed as it is being generated, it is possible to change simulation parameters in real time. For example, by middle clicking a node on the display and dragging, the user can dynamically change it's position. This will be immediately reflected in any calculations involving position. The second advantage comes from the ability to directly access the assorted objects representing parts of the simulation via the full set of python bindings generated to allow the Python module to interact with the compiled C++ code. This means that even data that would not generally be included in a trace file can be easily accessed, and displayed via plug-in. A good example of this is found in a plug-in that displays the full IPv4 routing table for any node in the network, which is included with PyViz.

PyViz has two significant drawbacks compared to NAM. The first is that, as it only displays the current state of the network, it is only possible to hold time steady or move it forward. Rewinding is not possible. The second, and much more limiting drawback is found in the mechanism by which PyViz parses input from trace sinks. It accumulates data for one tenth of a second of simulation time and displays an aggregate of that data. This is not an issue when viewing general data flow at real time, but it limits the ability to slow the visualization down to greatly. There is no viable way to see a single packet being transmitted, much less show it propagating across the radio channel of a wireless simulation.

The useful interface with the WiMAX NS-3 model is therefore composed primarily of two parts. Viewing overall throughput and wide-scale data flows is accomplished via

PyViz, but anything involving time resolution higher than one tenth of one second must be accomplished by examining trace files.

When animation is not sufficient or available to display some piece of data involved in the simulation, trace files are generally employed. NS-2 employs it's own custom trace-file format. Analysis using an NS-2 trace will generally involve creating code to manually pull the required information out of the trace file. NS-3 supports the generation of standard pcap trace files, which are used for analysis of real networks and are employed by many tools. This makes NS-3 potentially much more useful when trying to analyze network performance, as the vast array of tools available to analyze real networks are generally available.

Chapter 3

Overview of WiMAX

Consumers are increasingly using mobile devices such as smart phones and tablets to access the Internet while away from wired or relatively low-range Wireless Local Area Networks. This has led to demand for longer-range wireless networks that provide similar data rate to users. The Institute of Electrical and Electronics Engineers (IEEE) has therefore developed a family of Wireless Metropolitan Area Network (W-MAN) standards collectively referred to as IEEE 802.16 to deliver bandwidth at large ranges to highly mobile users.

The WiMAX Forum standardizes implementation requirements for IEEE 802.16 networks, and provides testing and certification of products to ensure interoperability [1].

3.1 OFDMA Physical Layer

Transmissions in WiMAX networks are organized into frames. Each frame consists of two subframes, one for downlink traffic (from the base station (BS) to the subscriber station (SS)), and the other for uplink traffic (from the SS to the BS). The frames are divided into these subframes using Time Division Duplexing (TDD), whereby the entire downlink subframe is transmitted then the entire uplink subframe is transmitted. There is a small amount of time separating the subframes. These gaps are known as the Receive Transition Gap (RTG) and the Transmit Transition Gap (TTG) [9]. The subframes may be equal length, or the downlink subframe may be longer, which is the standard configuration [9]. Figure 3.1 and Figure 3.2 show sample WiMAX frames.

All WiMAX networks use some variant of Orthogonal Frequency Division Multiplexing (OFDM). The general idea behind this form of modulation is that instead of modulating a single carrier spanning the entire available band (which, in the case of WiMAX ranges from 1.25MHz up to 20 MHz) [9], that large band is divided into many separate subcarriers (up to 2048). Adjacent subcarriers are chosen such that they are orthogonal to each other, and therefore a given subcarrier's signal will not interfere with any other subcarriers.

As each subcarrier is obviously granted only a small fraction of the total bandwidth, the maximum baud rate is substantially lower than what was available before the spectrum was divided. This effectively results in many relatively slow connections transmitted in parallel instead of a single faster connection, with similar total throughput. The actual overall modulation to be used in the final radio transmission is determined by running the set of outgoing symbols in a given time slot through an inverse Fast Fourier Transform (FFT). The receiving station then employs a standard FFT to decode the signal into the set of subcarriers.

In practice, subcarriers are grouped into subchannels. Allocation is then done in units of one symbol duration by one subchannel. Subchannels are generally not composed of adjacent subcarriers, and this type of grouping simplifies assigning a set of subcarriers spread over the spectrum. Even though in a standard OFDM implementation all subchannels, and therefore all subcarriers, are assigned to the same user at any given point in time, this system ensures that even if many of these subcarriers are not employed, transmission is still spread across the available band.

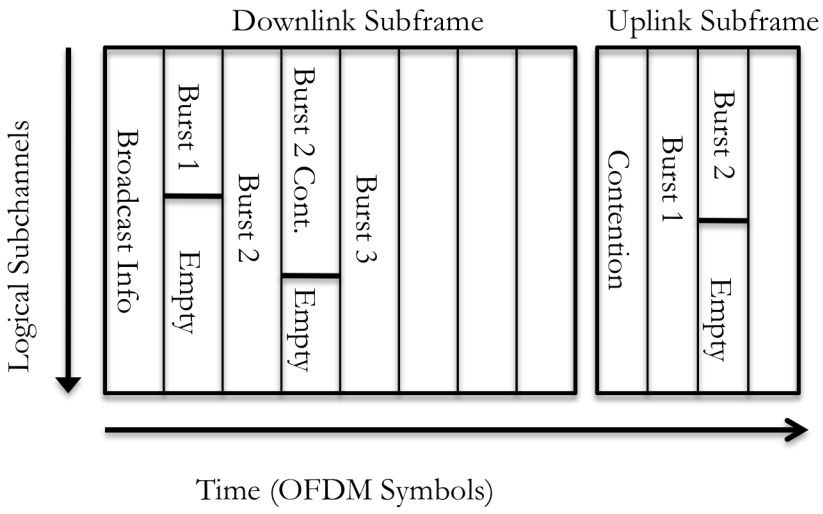

Figure 3.1 OFDM Frame

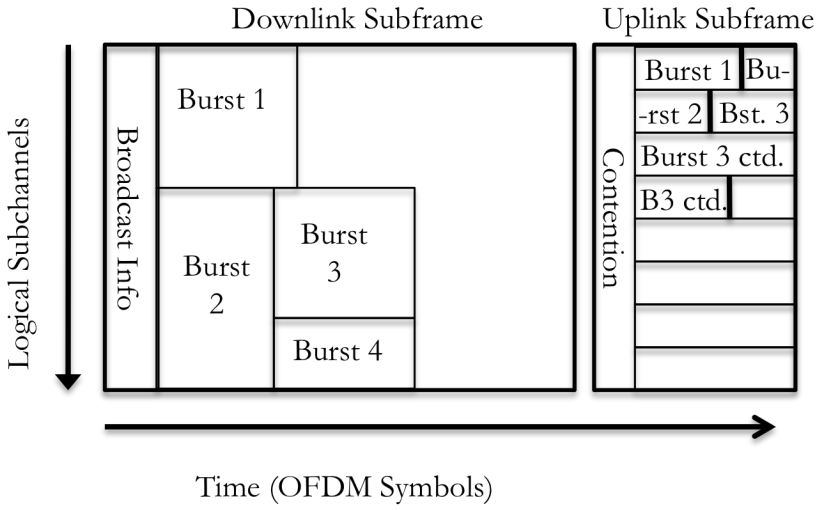

Figure 3.2 OFDMA Frame

The simple version of OFDM defined in the 802.16d standard grants all subcarriers to a single station at a given point in time (see Figure 3.1). This can result in substantial wasted bandwidth if that station only has a small amount of data to transmit and leaves most subcarriers vacant. This system brings several advantages over a simple modulation system. Notably, it can function well in situations involving interference over some subset of the subcarriers. Unaffected subcarriers continue to use higher modulation rates, while the affected subcarriers are either rendered unusable or simple

encoded at a lower rate. Also, since any given symbol (transmitted on a single subcarrier) has a relatively long transmission time compared to a simple modulation system, there is less potential for a symbol to interfere with subsequent symbols due to Doppler shift when mobility is an issue.

A natural extension of this idea is to avoid letting subcarriers go unused by granting only as many subchannels as are needed for a given transmission, and dividing the rest between one or more other users. This is referred to as Orthogonal Frequency Division Multiple Access (OFDMA) (see Figure 3.2). This variant is employed in IEEE 802.16e networks. A frame in an OFDMA network can be visualized as a two dimensional grid. Each block along the horizontal axis representing the time it takes to transmit a single signal, and each block along the vertical axis representing a subchannel. The specific implementation varies slightly depending on whether the communication occurs in the uplink or downlink subframe. If it is in the downlink subframe, a given transmission is assigned a rectangle on the grid. In the uplink subframe, transmissions are allocated as a series of blocks starting in the upper left corner of the grid, and filling in one row at a time from left to right.

3.2 Media Access Control Layer

WiMAX networks employ a scheduling system that only allows a given node to transmit when it has explicitly received a bandwidth grant from the Base Station. Much of the job of the MAC layer in a WiMAX network involves tracking the origin, destination, and purpose of packets. This information is used to request and grant bandwidth allocations.

3.2.1 Connections

Every entry in both the uplink and downlink subframe schedules in a WiMAX network has a connection identification number (CID) associated with it. A CID refers either to a specific connection between a BS and SS, or to one of several reserved, universal connections for initial network entry and broadcast packets.

During the Subscriber Station network entry and registration process, four connections are negotiated in pairs of uplink and downlink connections. These pairs are used for MAC layer signaling. The "Basic" connections are used for regularly scheduled packets that require exact timing, such as the Ranging packets. The "Primary" connections are used for other MAC layer traffic, including bandwidth requests.

3.2.2 Service Flows

Communication between the upper layers in WiMAX networks employs service flows. These are requested by either the SS or BS after full MAC layer connectivity is established [12]. Successful negotiation of a service flow results in two additional Connections be allocated for that pair of nodes. These are referred to as "Data" connections, and will only carry data from the upper layers and, in the case of the uplink connection, bandwidth requests. A Service Flow codifies a set of Quality of Service (QoS) parameters, most notably including bandwidth assurances or lack thereof.

3.2.3 Scheduling

Avoiding transmission collisions is an issue on any network, and many schemes have been employed to accomplish this. On wired networks where it can be reasonably assumed that all directly connected nodes can sense transmission from any other very quickly, a node can simply listen on the network and, if no transmissions are detected,

start transmitting. In the rare event that propagation time causes this to fail, both nodes wait a random period of time and try again.

In wireless networks this does not always work well, as it is possible that two nodes may be in range of the same Base Station but out of range of each other. This means that the base station will receive overlapping, unreadable transmissions from both, but neither will know the other is transmitting. Smaller scale wireless networks can overcome this reasonably well by having each node send a short Request To Send (RTS) to the base station, and waiting for permission in the form of a Clear to Send (CTS) message containing a duration before continuing. Since all nodes will receive the CTS message, and therefore know how long that node will be transmitting, none will attempt to transmit during that period. This means that the only packets that should ever run into collision problems are the very small RTS packets. However, this system does not hold up well under the large numbers of users found in WiMAX networks, as too many nodes would be competing for the contention-based RTS slots.

WiMAX networks expand upon the general concept of the RTS/CTS mechanism by creating explicit schedules with specific pre-assigned bandwidth grants. Bandwidth must be explicitly requested. The Base Station considers the bandwidth requests from all connected nodes, creates a global schedule that may or may not take specific Quality of Service (QoS) guarantees into account. Time is then organized into frames, each of which starts with the transmission by the base station of the new schedule. In most cases, once a node is connected, it can re-request bandwidth by adding a short message to the end of it's scheduled transmission period. However, in the event that a node does not have a bandwidth grant in a given frame, or if it is beginning the network entry process and is currently unknown to the Base Station, there is a short contention period in each frame that can be used to send un-scheduled transmissions.

Given a set of allocations, the problem of generating the final 2 dimensional mapping is non-trivial, and discussed farther in Chapter 5.

Chapter 4

NS-3 WiMAX Model

This Chapter provides a detailed description of the NS-3 based WiMAX simulation model developed as the majority of the work for this thesis. The first section details the important classes in the model, and the second discusses the flow of data through the simulator.

4.1 Class Structure

This section contains descriptions of significant classes in the model. Classes are grouped into several categories: the state machines, the classification system, the physical layer, the scheduling system, timers, headers, and several additional classes that do not fit into any of these categories. The descriptions will first provide a general view of what the class does, along with any other pertinent information about it, followed by a list of important or otherwise significant functions and data members, and brief descriptions of them. Accessors and similarly simple, straightforward functions will not be mentioned.

4.1.1 State Machines

Wimax2NetDevice

This is a template class for the Base Station and Subscriber Station NetDevices. It should never be directly instantiated. Any new subclasses of this NetDevice should implement transmit, receive, sendUp, the higher layer callback, tracing hooks, start_dlsubframe, start_ulsubframe, sendDown, init, and expire functions.

Subclasses serve as the state machines for the MAC layer of the type of node in question (either a BS or SS). As such, there are many data members of the superclass that hold important pieces of the simulation model, such as the connection manager, scheduler, classifier, and PHY layer. In the case of the scheduler, while a generic Wimax2Scheduler pointer is stored by the Wimax2NetDevice, subclasses actually store the location of a subclassed scheduler in this pointer. Significant functions implemented in this class are described in

Table 4.1 Wimax2NetDevice Functions

Function Name	Function Description
Send	Takes a packet from the upper layers, adds header information, calls sendDown from the subclass
Receive	This function is called by the subclass. It is the final function in the MAC layer before a packet is passed up to the higher layers
Classify	This function invokes the classification system to determine the appropriate CID given information about a packet's destination and what data it contains (MAC layer signaling information or data bound for the higher layers)
sendUp	This function receives a Packet object containing a chunk of a logical packet corresponding to what is transmitted in a single OFDMA allocation grid block, and stores it for later assembly.
TagIncomingMACPacket	This adds the appropriate MAC_TAG, designating which layer the packet is destined for, to a Packet received from another node based on the CID

	associated with it.
Assemble_incoming_packet	Based on the schedule either generated locally or received in a MAP packet, this concatenates the contents of the blocks comprising a single allocation to re-assemble a packet for processing
Process_received_packet	After a packet is assembled, this handles fragmentation and packing issues, as well as piggybacked bandwidth requests.

Wimax2BSNetDevice

This subclass of Wimax2NetDevice defines a Base Station's MAC layer. It manages connections to many Subscriber Stations, and decides if a given Subscriber Station will be allowed to connect to the network. It is also responsible for handling bandwidth requests, and uses a Wimax2BSScheduler to create scheduling information for each DL and UL subframes, and then transmits these schedules to all connected Subscriber Stations. It communicates with the upper layers using the Send and Receive functions defined in the superclass, and communicates with the PHY layer using the local transmit function and the superclass's sendUp function.

The class has two trace hooks, one when a packet is received from the upper layers for transmission, and another when a packet is about to be passed up to the upper layers.

Table 4.2 Wimax2BSNetDevice Functions

Function Name	Function Description
sendDown	This function takes a packet originating in the upper layers, adds necessary header data and enqueues it to the

	appropriate connection
transmit	Perform tracing functions and pass packet with correct, minimal set of headers to PHY layer
receive	Called after superclass's processing function completes. This either sends a packet to the upper layers or processes an incoming MAC-layer signaling packet
process_mac_packet	This checks the type field that starts every MAC frame, and sends the packet to the appropriate processing function
process_ranging_request	Used to process periodic or initial ranging requests, the first step of newtwork entry for a new SS. This allocates the Basic and Primary CIDs
process_bw_req	Processes a standalone (as opposed to piggybacked) bandwidth request.
process_reg_request	Allocates Seconday CID, and sends registration response, completing network entry for a new SS
Start_dlsubframe	Determines if DCD and UCD should be sent in this frame, and sets PHY to transmit mode
Start_ulsubframe	Puts PHY in receive mode and schedules the start of the next frame

Wimax2SSNetDevice

This class defines the state machine employed in Subscriber Stations. It handles automatic detection and entry into WiMAX networks, uses a Wimax2SSScheduler to generate and fill bursts for uplink transmission, and processes incoming downlink data. A description of important functions can be found in

Table 4.3 Wimax2SSNetDevice Functions

Function Name	Function Description
sendDown	This function takes a packet originating in the upper layers, adds necessary header data and enqueues it to the outgoing data connection for this SS.
transmit	Perform tracing functions and pass packet with correct, minimal set of headers to PHY layer.
expire	Called when several types of timer expire, including DL and ULMapTimers to ensure synchronization, T3 and T6 timers to handle ranging and registration request timeouts, respectively.
receive	Called with a packet is fully assembled, after superclass has processed fragmentation and packing. Sends into the MAC frame processing system or to upper layers, depending on CID.
process_mac_packet	This checks the type field that starts every MAC frame, and sends the packet to the appropriate processing function
process_FCH	Called at the start of every DL-Subframe. Assembles Frame Control Header, which defines the DL-MAP's allocation and always occupies the same allocation, and the DL-MAP
process_dl_map	Uses information in DL-MAP to determine which downlink packets are being sent to this node (including broadcast packets), and schedules assembly.
process_ul_map	Uses information in UL-MAP to construct the data structures used by the SS Scheduler class for creating the UL Allocations

process_dcd	Processes Downlink Channel Descriptor packet, which defines the meaning of the Downlink Interval Usage Code (DIUC) used to specify encoding for DL Packets. Necessary early in network entry.
process_ucd	Processes Uplink Channel Descriptor packet, which defines the meaning of the Uplink Interval Usage Code (UIUC) used to specify encoding for UL Packets. Necessary early in network entry.
process_ranging_rsp	Processes a response to either an initial or periodic ranging request, adjusting transmission power if necessary and an essential step in network entry.
process_reg_rsp	Completes network entry.

4.1.2 Classification System

Both Base and Subscriber Stations track which nodes they are connected to, and maintain several connections with each. They also maintain systems for determining which of these connections should be used for a given packet, and for traffic originating from the upper layers they employ service flows to perform QoS operations.

Connection

An instance of this class represents a Layer 2 connection between nodes. A pair of connected nodes (one BS and one SS) will have several connections between them that are used for different purposes. Each connection has a sixteen-bit identification number that is unique within the network referred to as a Connection IDentifier (CID). Connections also maintain a queue of packets waiting for transmission. Note that it is actually a deque, as occasional time sensitive messages must be put on the front of the

queue. Almost all connections fall into one of four categories: Basic, Primary, Secondary, or Data. A handful of others that have very restricted uses, such as an initial ranging and broadcast, also exist. Significant functions of the Connection class are listed in Table 4.4

Table 4.4 Connection Functions

Function Name	Function Description
Enqueue	Adds a packet to the internal connection transmission queue
Enqueue_head	Adds a packet to the front of the internal connection queue – used to reinsert remaining fragments
Dequeue	Removes the packet at the front of the queue for transmission or processing

ConnectionManager

This class is used to manage the various Connections associated with a particular Station. It maintains lists of both incoming and outgoing connections, and can find a Connection based on the CID and direction (in or out of the current station).

SDUClassifier and destClassifier

SDUClassifier is an abstract base class from which classification systems can be derived. The classifier's job is to associate MAC addresses with CIDs.

DestClassifier is a simple subclass of SDUClassifier that only uses the origin of the packet (Layer 2 vs higher layers) and the destination MAC Address. Unlike some potentially elaborate classification systems, there is only one instance of this class per Base Station or Subscriber Station, and so long as a more elaborate classification system isn't developed and inserted at higher priority, the Classifier List will consist of only one element.

PeerNode

The PeerNode serves as an index for the Connection objects associated with a single other station. A Subscriber Station will only have one PeerNode, for the Base Station it is associated with. A Base Station will have one PeerNode for each Subscriber Station associated with it. A single PeerNode will contain six connections in most cases. These are three pairs of outgoing and incoming Connections, one pair represents the basic connections, one the primary connections, and the last represents the data connections.

4.1.3 Physical Layer and Channel

Wimax2Phy

This class represents the PHY layer of either a BS or SS. It is responsible for breaking a packet into small pieces corresponding to a specific block of one subchannel by one symbol-duration on the OFDMA allocation grid, scheduling calls to the channel's Send function, and receiving single blocks from the channel for caching and eventual assembly in the MAC layer.

It also contains many functions that provide information about the size of the allocation grid, slots, and individual blocks in the allocation grid.

Wimax2Channel

This class represents that physical channel over which radio transmissions are sent. It expects to use the built-in ns3 defined COST-231 propagation model, but can accept other models that are children of the ns3::PropagationModel class. It sends every transmission to every node except the one that is transmitting, though many of these nodes will ignore the incoming data. The channel model simulates propagation loss and propagation time.

4.1.4 Scheduling

Wimax2Scheduler

This class serves as a base class for the BS and SS Scheduling classes. In the NS-2 model, it contained the functions responsible for filling bursts and managing packing and fragmentation. However, due to these functions being much simpler to implement using NS-3's packet architecture, and the number of conditionals required to insert bandwidth requests only for uplink traffic, the relevant functions are now located in the subclasses.

Wimax2BSScheduler

This class is responsible, via the Schedule function, for allocating bandwidth in both the uplink and downlink subframes. It does this based on data in local queues for the downlink subframe, and from received bandwidth requests in the uplink subframe. These schedules are transmitted to Subscriber Stations via the UL_MAP and DL_MAP messages that it schedules for the start of every frame. It is also responsible for allocating and filling bursts for outgoing transmission from the Base Station that owns this scheduler using the transfer_packets_for_dl_subframe function. This function handles packing and fragmentation.

Wimax2SSScheduler

This class is responsible for examining the allocations granted in the uplink subframe, and creating bursts for any with CIDs corresponding to the Subscriber Station that owns this scheduler. If an allocation is found, it determines the maximum size based on the Uplink Interval Usage Code (UIUC) and duration, and transfers the appropriate amount of data out of the Connection's queue and into the newly created burst using the transfer_packets_for_ulsubframe function. This function handles packing, fragmentation, and adding piggybacked bandwidth requests as appropriate.

Burst

A burst object is created by a Scheduler to define an allocation. They correspond to specific regions on the OFDMA allocation grid, and have a maximum byte allocation. Because they are associated with specific allocations, they are also associated with specific connections. Functions in the scheduler subclass are used to transfer packets out of the queues in Connection objects and into a Burst. While the Burst class supports multiple packets being enqueued, this functionality is not used in the current version of the simulator. It was used extensively in NS-2, as storing multiple distinct Packet objects was necessary for simulating packing. As discussed in Section 2.3 Packets, NS-3's packet architecture allows packing to be easily performed in a single Packet object. Therefore, only one packet will be enqueued to a given burst.

Profile

This class defines serves to link a specific modulation rate with a specific Interval Usage Code (IUC). It is created in the subscriber stations based on the contents of the Downlink Channel Descriptor (DCD) and Uplink Channel Descriptor (UCD) messages sent periodically.

Subframe

This class manages information associated with a given subframe in a specific node. It stores the profiles, and stores data scheduled to be transmitted in the frame in the form of PhyPdu objects.

Framemap

This class is responsible for storing subframe objects for the current UL and DL subframes, parsing received Downlink-Map (DL-MAP), Uplink Map (UL-MAP),

Uplink Channel Descriptor (UCD), and Downlink Channel Descriptor (DCD) frames, and generating those messages for transmission.

4.1.5 Timers

Timers are used to schedule events when that event may need to be canceled. They effectively serve as wrappers for the Scheduler's management system. When a timer is started, the event in question is scheduled, and its unique EventId is stored. If the timer is paused or stopped, this is used to cancel the scheduled operation. Otherwise it executes as any normally scheduled function call would and the EventId is discarded upon completion.

Wimax2Timer

The WiMAX specification defines many operations that must be completed at regular intervals. The Wimax2Timer class is subclassed for many of these, with appropriate function calls to complete the required tasks.

DlSubframeTimer and UlSubframeTimer

These expire at the beginning of a new Subframe. They are responsible for constructing the PHY_INFO_Headers detailing which OFDMA allocation blocks will be employed as part of the allocation being used, and scheduling calls to the transmit function for packets being transmitted in the appropriate subframe.

4.1.6 Headers

The 802_16headers.h file contains definitions for 35 classes encapsulating the assorted header and frame definitions found in the 802_16pkt.h file in the original NS-2

simulation. These classes are listed in Table 4.5 Headers. The original structures are preserved for the sake of making integration of the old code with the new packet and header formats simpler. There are five classes defined in this file that do not have direct counterparts in the NS-2 model.

The first is the All_Headers class, which serves to keep track of all relevant header data within a given station. It is simply an encapsulation of assorted headers and some metadata regarding a packet, and is stripped off prior to transmission (and regenerated upon receipt), being replaced with only the appropriate set of headers for that particular packet.

The second is the MISC_OTHER_HEADER class, which simply serves to make serializing and deserializing All_Headers instances somewhat simpler. It should never be instantiated outside the All_Headers class's functions.

The third is the Type_Check class. This is used to examine incoming MAC layer packets to determine which frame type the packet contains based on the one byte type field found at the beginning of every MAC frame. In the NS-2 model, the pointer to the data portion of the packet containing the frame in question could simply be cast to the structure representing any frame type, and so long as that was the only member accessed there was no danger of mismatching frame sizes and causing a Segmentation Fault. However, in NS-3 where in order to access the contents of a packet it must be deserialized, we must be careful not to try to deserialize a frame from the packet that would pull more data from the packet than it contains. Therefore, we use the Type_Check class with the PeekHeader function to only deserialize the first byte corresponding to this field, then choose the correct frame type to deserialize based on the contents.

The fourth class is the MAC_TAG class. This is not a header that is serialized into a packet's data buffer, but rather a piece of meta data applied to the packet. This is used

to track whether the packet was passed down from a higher layer or generated by the MAC layer. This is stripped before transmission in the interest of realism, as in an actual network this metadata could obviously not be transmitted without taking any room in the stream of bytes composing the packet. On receipt of a packet from another station, a new copy of this tag is generated based on which connection ID is listed on the packet's header.

The final class is the UL_Protocol_Header class, which stores the 16-bit protocol field used by higher layers in NS-3 to classify traffic into the correct application. This parameter must be preserved from the input to the MAC layer in the transmitting node to the argument in the call to the upper layers in the receiving station.

As all of these classes save MAC_TAG are subclasses of the Header class included as part of the core simulator, they all have the same set of significant functions. The MAC_TAG class has the same set of functions, as the Tag and Header classes are similar.

Table 4.5 Headers

Name	Description
Frame_Prefix_Header	Stores information about DL_MAP allocation. Always sent at start of Dl-Subframe
Generic_MAC_Header	Also called GMH, first 6 bytes of almost every packet transmitted. Stores CID, length, and flags for packing and fragmentation
Signaling_MAC_Header	An alternative to the GMH used for bandwidth requests in a granted allocation
RANGING_RESPONSE_ FRAME	Stores a Ranging Response

Name	Description
DL_MAP_IE	A single entry in the DL_MAP known as an Information Element (IE)
Fast_Ranging_IE	
UL_MAP_IE	A single entry in the UL_MAP known as an Information Element (IE)
CDMA_MAP_IE	Extension to UL_MAP_IE for CDMA entries with top and code
DCD_FRAME_HEADER	Stores a DCD
DL_MAP_FRAME_ HEADER	Stores a DL_MAP
DSA_REQ_FRAME_HEADER	Stores a Dynamic Service Flow Request
DSA_RSP_FRAME_HEADER	Stores a Dynamic Service Flow Response
DSA_ACK_FRAME_HEADER	Stores a Dynamic Service Flow Ack
UCD_FRAME_DATA	Stores a UCD
UL_MAP_FRAME_HEADER	Stores a UL_MAP
REGISTRATION_ RESPONSE_FRAME	Stores a Registration Response – receipt of this completes network entry
REGISTRATION_ REQUEST_FRAME	Stores a Registration Request
RANGING_REQUEST_FRAME	Stores a ranging request to be transmitted in an allocated (non-contention) slot

Name	Description
CDMA_REQUEST_HEADER	Alternate MAC Header for transmission in contention slots for Initial Ranging or Bandwidth Requests
PHY_INFO_Header	Stores information about which blocks are included in burst transmission
FRAG_SUBHEADER	Stores a Fragmentation Subheader – not used when both Fragmentation and Packing are enabled
GRANT_MAP_SUBHEADER	Stores a Grant Map Subheader, used for piggybacked bandwidth requests
PACKING_SUBHEADER	Stores a Packing Subheader – used when both Fragmentation and Packing are enabled
FFB_SUBHEADER	Stores a Fast Feedback Subheader
ARQ_FB_IE_HEADER	Stores an ARQ feedback information element
mac802_16_mob_scn_req_frame	SS Scanning request frame. Multiple BS can respond to this to decide on Handover
mac802_16_mob_scn_rsp_frame	BS Scanning response frame. One SS may receive more than one of these per request.
NEIGHBOR_ ADVERTISEMENT_ FRAME_HEADER	Stores a BS->BS neighbor advertisement frame
MOB_HO_IND_ FRAME_HEADER	SS Handover indicator frame

Name	Description
BSHO_RESPONSE_ FRAME_HEADER	BS Handover Response Frame
MSHO_REQUEST_ FRAME_HEADER	SS Handover Request Frame
MISC_OTHER_HEADER	Stores a subset of the data in All_Headers to make Serialization and Deserializeation cleaner
Type_Check	A single byte, used to check the type of received MAC packets. The first byte of the payload of these packets is always a type code
UL_Protocol_Header	Stores the unsigned 16 bit integer used by higher layers to represent the protocol for the packet
All_Headers	Large aggregated meta-header used internally to conveniently store many potential headers. It should never be on a packet that is being transmitted.

GetSerializedSize

This returns the total size that will be written to or read from a Packet's byte buffer if this packet is serialized or deserialized. For simple headers this is generally simply a return [constant] statement. For frames that contain a variable number of information elements or TLV encoded data, some calculation may be necessary. This is called by NS-3 when Serializing a Header to a packet to expand the byte buffer by the appropriate amount.

Serialize

This writes the contents of the Header to the byte buffer in a Packet. This is done by a set of functions that write integers or unsigned integers of varying sizes (correcting for network byte order when necessary). Note that this means everything must be written in full bytes. Since the layout of a given header or frame structure will generally use the minimum number of bits necessary to store a given piece of information, many of these functions employ bitwise operations to either pack several members of a structure into one larger temporary variable to be written or to fragment large members of a structure for writing across multiple temporary variables.

Deserialize

This is the inverse of the Serialize function. It reads data from a packet's byte buffer and stores that data in the appropriate members of the data structure representing the Header that had been Serialized to the packet previously. Similarly to Serialize, the functions used to read data from the byte buffer only work in one or more full bytes, so for structures that do not map cleanly to bytes, temporary variables and bitwise operators are used to unpack the data correctly. The value returned is the total number of bytes removed from the packet's byte buffer, which is generally accomplished in this model by simply calling the GetSerializedSize function, but can also be accessed from the Buffer::Iterator used to pull the data out of the buffer.

4.1.7 Miscellaneous Classes

Wimax2Common

This file does define a small class, but the main purpose of this file is to provide a place where certain commonly referenced enumerations and define commands can be placed without risking circular dependencies. The most notable definitions found in this file are debug2 and debug10 to printf, and "NOW" to the current simulation time in

seconds. Significant enumerations include those defining the modulation rate, and direction of traffic.

Two simple functions are defined here. The first gets a string representation of a Mac48Address, and the second returns a c-string representation of the type of MAC header included with a packet. Both of these are commonly used for printing debugging information.

Mac802_16MIB

The MAC MIB, or MAC Management Information Base is defined as a class with many public simple data members. It is passed from the simulation script to specify a large number of simulation parameters. It is used because passing a single object around is much easier than passing all 48 members independently. Also, the constructor allows a simple way to set default values for these parameters that are still very easy to change in the script. It is passed into the simulation by calling setMacMIB on each NetDevice when it is created.

PHY_MIB

Similarly to the MAC MIB, the PHY MIB, or PHY Management Information Base contains several simple data members representing simulation parameters. They are passed in through a single object rather than individually because it is easier to pass a single object than 9 independent parameters. It is passed into the simulation by calling setPhyMIB on each NetDevice. The NetDevices then pass the information stored in the class down to their PHY layers.

4.2 Data Flow

A network is, basically, a system for passing data from one node to another. In that light it is useful to look at the WiMAX Network model as a path along which packets are moved. This section serves to illustrate the path a packet will take through the class structure from the time it enters the model until the time it is either passed up to higher layers or processed and discarded. Each subsection describes the path for a particular segment of the model.

4.2.1 Subscriber Station Send Procedure

A graphical representation of this procedure can be found in Figure 4.1. Data originating from a subscriber station, either from the higher layers or from Layer 2 for management purposes, is initially added to a specific connection's outgoing packet queue. A subscriber station will generally have four outgoing connections, two for management messages of varying priority, one for data from the higher layers, and one that is used by every node during initial ranging and network entry. The origin of the specific packet from either the higher layers or the MAC layer determines which connection is used to send it.

Once a packet is enqueued, it sits in the queue until the subscriber staiton's scheduler finds an allocation for that specific connection in the UL_MAP for a particular frame. When this happens, the scheduler transfers packets out of the connection queue and into a burst of the specific size granted in the UL_MAP. In the NS-3 version of the simulator, a burst will usually only contain a single packet in its queue. This is because, as described in Chapter 2, NS-3's packet structure allows for much greater simplicity in packing and fragmentation, so a queue of separate packets is unnecessary. The packet in the burst will have all headers properly in place, including packing and fragmentation subheaders, and a single Generic or Signaling MAC Header at the start.

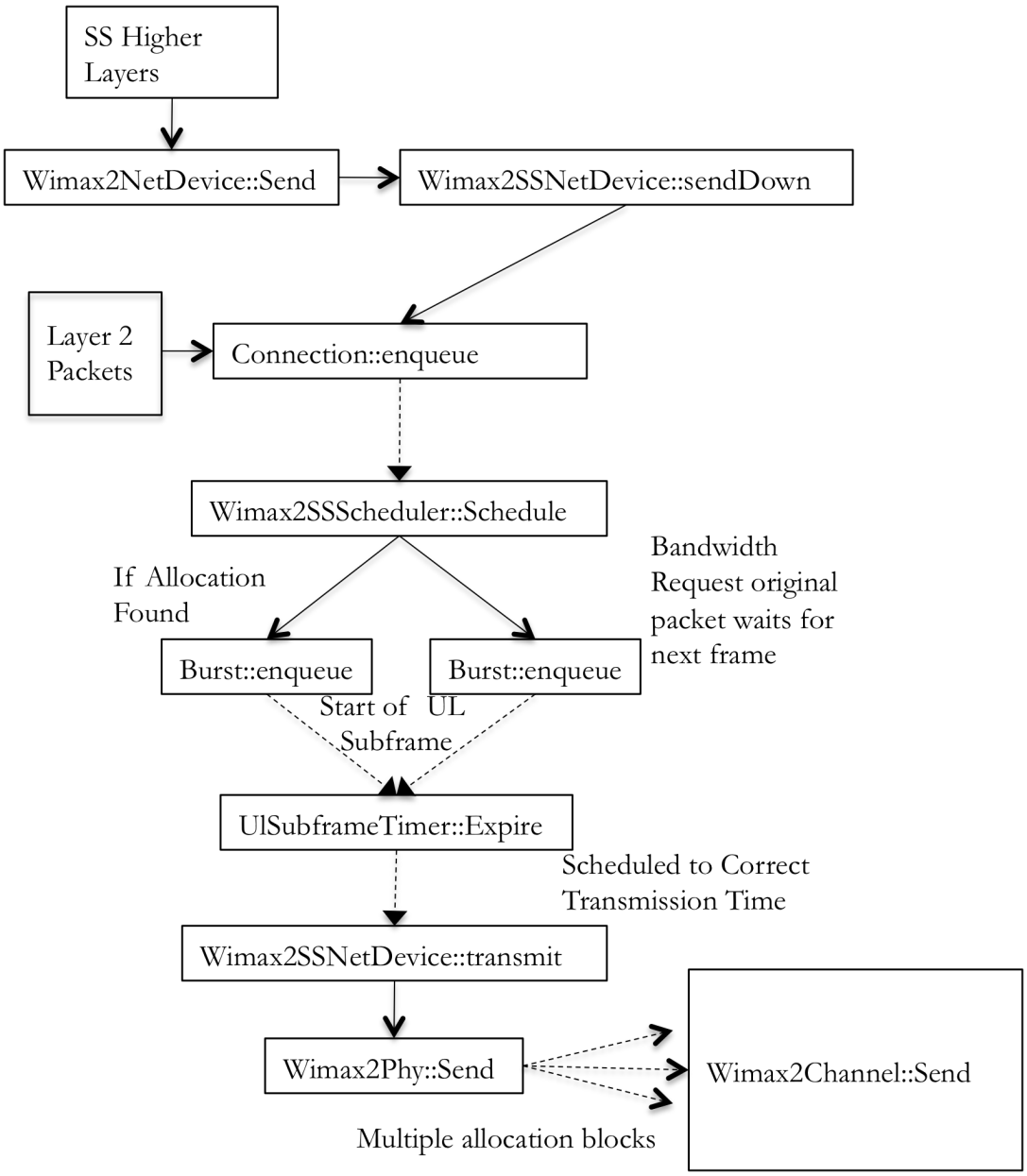

Figure 4.1 SS->BS Transmit Procedure

The now fully assembled packet sits in the burst until the start of the UL Subframe, at which point the UlSubframeTimer associated with that particular node expires. The expiration function on that timer schedules the transmission of each burst based on the transmission time determined by the scheduler. The transmission function in turn calls the PHY layer's send function, which breaks the assembled packet into small blocks,

each corresponding to a single allocation block of one subchannel by one symbol in the OFDMA allocation grid. It then schedules each block to be transmitted at the appropriate time based on the block's x-axis value in the OFDMA allocation space.

4.2.2 Base Station Send Procedure

A graphical representation of this procedure can be found in Figure 4.2. It is very similar to the Subscriber Station send procedure with a few slight differences. First, selecting the appropriate connection for a packet is somewhat more difficult. This is because instead of there being only four connections as described above, there are a set of broadcast connections, and then data and MAC signaling connections for each connected subscriber station. The other primary difference is found in the lack of a need to send bandwidth requests. This is because the Base Station performs scheduling locally, and therefore can directly examine the amount of data sitting in each connection's queue. If an allocation is not granted, no explicit request must be made, as the scheduler will directly observe the length of the queue when the next subframe schedule is generated, and grant an allocation if possible.

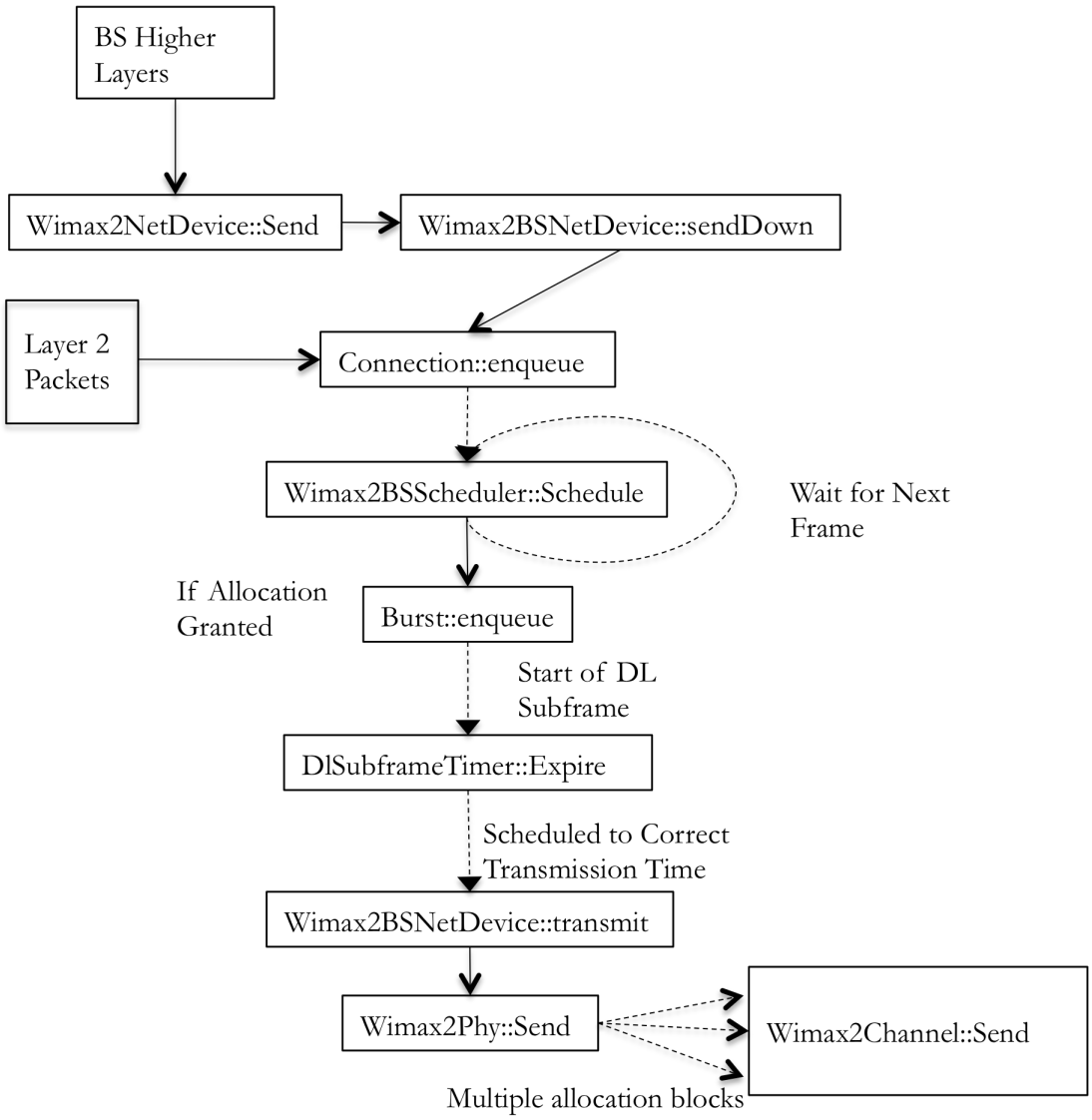

Figure 4.2 BS->SS Transmit Procedure

4.2.3 Receive Procedure for Base Station and Subscriber Station

A graphical representation of this procedure is found in Figure 4.3. The procedure is effectively identical for both Base Stations and Subscriber Stations, as the same functions are called in each subclass. What varies between them is, of course, the

contents of several of these functions. Most notably, process_mac_packet handles a different set of messages for each.

As each individually transmitted allocation block is received at the appropriate time, they are stored for the remainder of the frame in a grid with dimensions corresponding to the OFDMA Allocation Grid. This grid also stores information about whether a collision occurred at that block, or if there was any other kind of reception error.

Based on the contents of the subframe schedules, which are transmitted to the Subscriber Stations in the same position in every frame, facilitating their decoding without the need for the map to have already arrived, each node knows when any relevant packets may arrive, and on what specific allocation blocks they will be transmitted. This allows a call to a packet assembly function to be scheduled for just after the last block is received. The assemble_received_packet function examines each block that should have been received. If all have been received and none suffered errors or collision, then the packet is reassembled. As soon as reassembly is complete, the packet is checked for fragmentation or packing, with each being handled appropriately by enqueueing the fragment to the Connection object or sending the unpacked, complete packet to the reception system. In the case of fragmentation, once the last fragment is received, the assembled packet is sent up in the same manner. From there the packet is either sent into the functions that decode and process Layer 2 signaling packets or passed up to the higher layers, as appropriate.

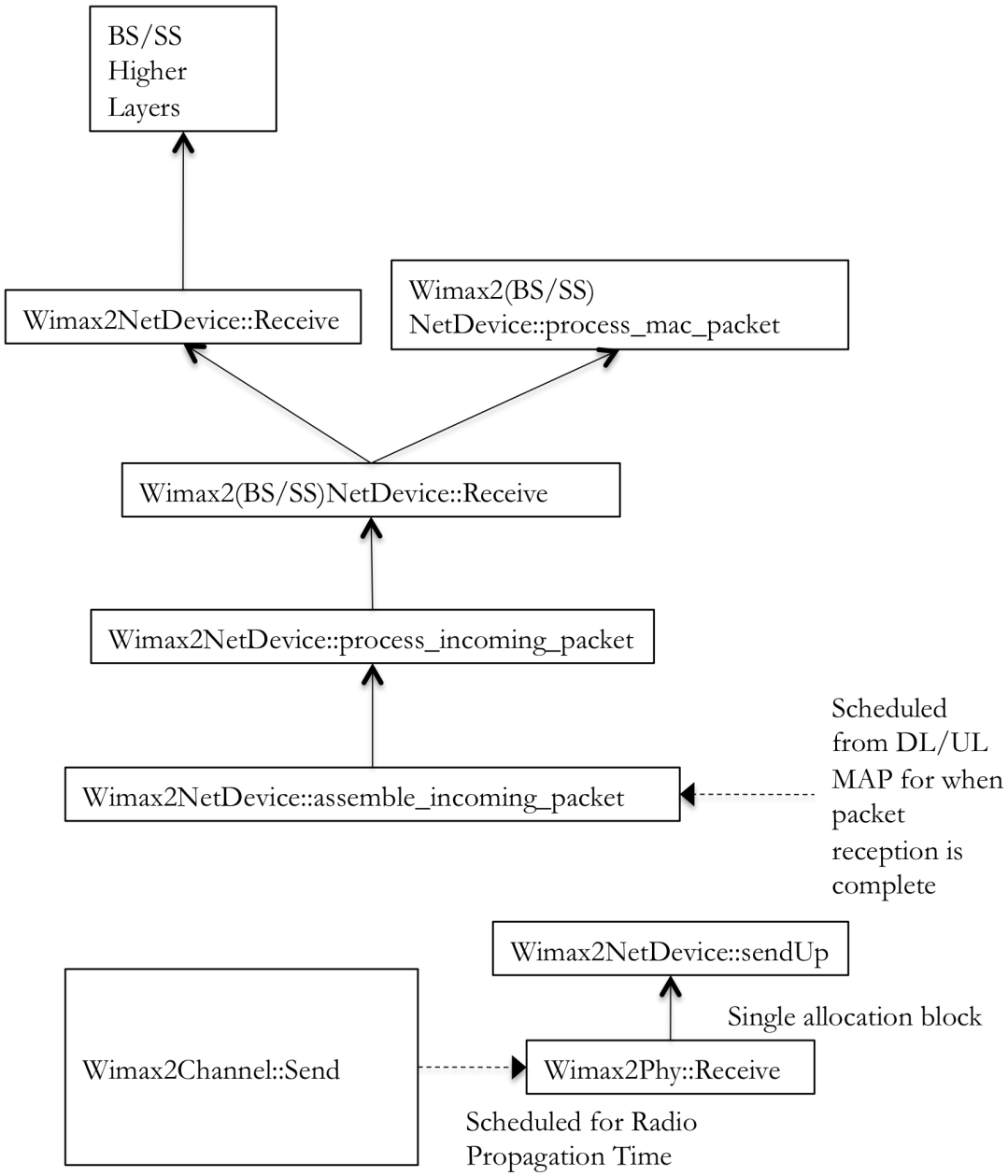

Figure 4.3 Receive Procedure

4.2.4 Network Entry Procedure

Due to the fact that several connections, a service flow with a QoS profile, and a variety of ranging information must be determined before regular higher layer communication can proceed, the network entry procedure is somewhat involved. It spans a total of 9 frames for each Subscriber Station, though these may overlap entirely if multiple stations try to connect at the same time and do not choose the same random contention slot for early messages.

A summary of which station sends which message at which time can be found in Table 4.6. This table also includes information about the progression of states the SS passes through between being created and being ready to accept communication from the higher layers.

Table 4.6 Network Entry Procedure

Frame Number	Base Station Activity	Subscriber Station Activity	Subscriber Station State Machine State
1	Send DCD and UCD	Send Contention-based Ranging Request Frame	DISCONNECTED SYNC_DCD SYNC_UCD RANGING WAIT_ANON_ RNG_RSP
2	Send Ranging Response – Allocated Basic and Primary CID	BW-REQ for Ranging Request using new CID	RNG_ALLOC
3	Grant Allocation for Ranging Request	Send Ranging Request using new PRMARY CID	WAIT_RNG_RSP
4	Send Ranging Response	Generate Registration Request and send Bandwidth Request for it	REGISTER

5	Grant Allocation for BW-REQ	Send Registration Request	
6	Send Registration Response		CONNECTED
7		Bandwidth Request for Dynamic Service Addition (DSA_REQ) Request	
8	Grant Allocation for DSA_REQ	Send DSA_REQ	
9	Send DSA_RSP and grant allocation for DSA_ACK	Send DSA_ACK	

Chapter 5

Key Issues in Translation and Improvement

This Chapter describes the significant challenges and required changes needed to move from the NS-2 model to the NS-3 model. It includes both issues involving replicating the functionality of the original model and changes made to add new functionality.

5.1 Packet and Header Differences

The single most significant and time-consuming issue centered on the differences in the packet and header architecture in NS-3 and NS-2. An overview of the differences can be found in Section 1.3. While the changes seem as if they would be fairly straightforward, a number of issues complicated the matter, and the frequency with which Packet objects are used in the model made this a very significant issue.

One major problem is that, in NS-2, when a header is accessed, it is modified in-place in the packet. In NS-3, a header must be removed and added back later for any changes to persist outside the immediate scope. Essentially, almost every path through the code had to be examined, with headers needing to be removed and replaced before and after most function calls using a packet. Considering the size of the model and the number of functions that take packets as arguments, this was very time consuming to retrofit.

Even once all the additions and subtractions of headers were added, there was the issue that much of the extant code assumes access to the entire set of potential headers. Since code was written assuming easy access to every header, especially the GMH and PHY_INFO_Headers, it made sense to find a way to keep these easily accessible

without needing to explicitly add and subtract both of them in every function. The All_Headers class is used to solve this problem. It contains effectively the same set of headers that the NS-2 packet structure includes, so within a given node, the remaining NS-2 code can function reasonably well. It is stripped off before transmission and only the correct, minimal subset of headers is used in the transmitted assembly. This means that for transmission purposes we still get the benefit of having the actual exact set of bytes to be transmitted, fragmented, or packed with other packets, but for processing tasks internal to the node we can safely assume access to any and all headers.

5.1.1 Fragmentation and Packing

One of the chief advantages of the NS-3 packet architecture is found just prior to and just after transmission. Packets are transferred to and from bursts, and packing and fragmentation processing occurs.

Due to the fact that in NS-2, there is exactly one copy of each header attached to every packet, some tasks are extremely awkward. For example, WiMAX stations can optionally employ a packing system in which several logical packets are packed into one transmission. This is specified by a flag in the Generic MAC Header, and then each packet is preceded by a small subheader giving the duration of the following segment. Because the Packing Subheader must be included multiple times in a single transmission, NS-2 cannot easily assemble individual packets for transmission when packing should be employed. This leads to the necessity of managing lists of packets that must be transmitted together. Each of these packets will have a Packing Subheader associated with it, as it should, but also a full set of other headers including a Generic MAC Header with it's length field, and an NS-2 common header with yet a third length field. These fields will generally not all agree, due to each potentially assuming different headers are present in addition to the payload. This leads to some confusion, and multiple calculations for the total amount of data contained in a burst.

NS-3's model by which all packet objects are simply a string of bytes makes packing multiple data units into a single Packet object straightforward. It is extremely simple to copy data from one packet onto the end of the buffer in another, or to create a new packet containing a specific subset of the data in an original packet. Because individual packets being packed into a burst do not have their own Generic MAC Headers, and the packing subheader's length field is set based on the current byte length of the packet's buffer, there is no ambiguity over size.

Because the packing and fragmentation works in a completely different way, the functions used to do this processing are completely new in the NS-3 model. The NS-2 version of this function, Wimax2Scheduler::transfer_packets1, is nearly 800 lines of code long (with very few of those lines being comments). The NS-3 version, which performs exactly the same tasks, is around half the length with extensive commenting. The NS-3 version generates a burst of completely unambiguous size, and the general process is extremely intuitive. Packing is accomplished by copying the contents of the packet buffer and adding it to the end of another packet, while fragmentation copies regions out and deletes them.

5.2 Python Bindings Generation

NS-3 has optional support for scripting in Python in addition to C++, which is required to use the PyViz visualization module included with NS-3. Support for python is facilitated through the use of compiled binding files mapping a portion of the C++ API. While the code from which these are compiled can be generated by hand, it is impractical for a model as large as this simulation. A tool known as PyBindGen is included with NS-3 that tries to automatically generate binding files by scanning the C++ API, and it works reasonably well but there are issues.

The most notable issue is that only a subset of C++ is actually supported. Most current features are well supported, but several prominent exceptions were found, mostly centering around the older C-style coding practices which are relatively common in the NS-2 model.

Most of these do not cause issues. While it is uncommon to see this style of coding in newer C++-based libraries, it is all still legal in the language. However, Support is notably lacking for anonymous types, functions generated by macro such as the <sys/queue.h> linked list modules, and double pointers. The need to avoid using these led to a substantial portion of the code base being rewritten to employ newer alternatives, mostly from the C++ Standard Template Library.

The other large issue centering on this packing is the lack of useful output. Many errors are generated both for new code and the existing NS-3 API, some of which can be safely ignored and some of which can't. There is little to no documentation on which type of error is which. Even finding errors can be time consuming, as they rarely stop scanning, and may not appear until attempting to compile the bindings, or run a simulation using them. This makes retrofitting code into the subset of C++ that can be properly scanned using this tool extremely time consuming as it involves rescanning and recompiling large portions of the simulator repeatedly, while relying on vague error reports which may or may not actually cause problems in the simulation.

5.3 OFDMA

The NS-2 model implements the OFDM PHY layer defined in the 802.16 standard instead of the OFDMA PHY layer. As discussed in Section 2.1, there are certain drawbacks to this compared to the OFDMA system implemented in the NS-3 model. This is especially noticeable when dealing with small packets, such as those used in the Layer 2 Network entry procedure described in Section 3.2.4. Most of these messages are around 50 Bytes or less, but each message is still granted use of every subchannel at

a given time slot. Depending on the allocation used, this can result in extremely significant (greater than 90%) of the allocation being unused.

Based on a number of notes in the code, it seems clear that the NS-2 model's developers were in the process of implementing an OFDMA PHY layer, but it was a distinctly unfinished task. Changing this required an overhaul of the Base Station and Subscriber Station schedulers, several changes to the UL- and DL- MAP information elements, and a redesigned transmission and collision detection system.

The NS-2 model uses a system in which a single packet can be received at a time. Receiving parts of two packets at once will cause the simulator to assume a collision and drop both. However, in OFDMA allocations, multiple simultaneous receptions are entirely allowed, and frequently specifically scheduled for. Therefore, a scheme in which data from multiple sources can be received simultaneously is necessary.

For each frame, the NS-3 model builds a grid of received blocks. Each time a node receives a block of one subchannel by one OFDM Symbol, that block is stored in the appropriate location in the grid. This allows a node to easily detect collisions on a specific block in the allocation grid, and also allows a more realistic simulation of OFDMA transmission. Packets are actually split into many pieces sized appropriately to fill one block with the assigned modulation, and pieced together later based on the regions defined by the schedule.

Chapter 6

Sample Application of NS-3 Model

During the scheduling process, most of the steps are relatively simple. However, the process of taking a set of allocation sizes and mapping them to rectangular allocations on a two dimensional grid as is required in the downlink subframe is very difficult to do in an ideal fashion. In fact, it is a relatively straightforward variation on the bin packing problem [3], which is known to be NP-Complete. This means that the only known method for determining a solution guaranteed to be ideal is to generate every possible solution and check each one. This process scales exponentially with the number of potential solutions, making it impractical to rely on for a process that is required to be reliably completed for every frame, each of which lasts only 0.005 seconds.

Because of this impracticality, several heuristic algorithms have been devised that generate reasonably good mappings quickly. These heuristics can be objectively evaluated by comparing the number of unallocated and over allocated slots to the ideal solution found using the exponential time full-search algorithm. Two related algorithms that provide good approximations to the ideal mapping with much faster run times are known as the One Column Striping with non-increasing Area first mapping (OCSA) [3], and enhanced OCSA, a modification of that algorithm. After discussing these, a new modification will be presented and then analyzed using data gained from the NS-3 simulation model.

All modeling and examples, unless otherwise noted, will be performed on an allocation grid of 14 slots by 30 subchannels. This assumes that a single slot in the downlink subframe is two symbol-times in duration, that the total downlink subframe size is 33 symbol-times by 30 subchannels, and that the first 5 symbol columns are used for the preamble, the FCH/DL_MAP, and the UL_MAP and the DCD and UCD if they are

transmitted in this frame. These are scheduled as such because failing to schedule any of these can result in serious problems, including subscriber stations assuming they have lost the connection and exiting the network.

To quantify the results of these algorithms compared to the ideal mapping, both are generated and the following two parameters are calculated for each. First, the number of allocation blocks that are allocated to a burst, but provide bandwidth past what is needed to fully transmit the burst. Second, the number of allocation blocks that are not allocated to any burst, when one or more bursts could not be transmitted due to allocation space in large enough blocks not being available. The relative quality of mappings produced by a heuristic is determined by comparing normalized averages of these parameters.

6.1 OCSA

The basic goal of this heuristic is to enumerate all possible allowed (not larger than the bounds of the subframe) sizes to map a given burst, then position the minimum-waste options in descending size order, while packing smaller packets in to leftover space. The more specific description is that after enumerating the possible mappings, and sorting the allocations in descending order by size, each allocation starting with the largest is mapped using it's lowest-waste mapping, starting from the bottom-right hand side of the allocation grid. After an allocation is mapped, if there is any space left above it, all remaining allocations are checked to see if any will fit into the leftover space. Note that to ensure that future allocations will not be height limited by this mapping, allocations mapped to the leftover regions here cannot exceed the width of the initial mapping below it. This process is repeated, stacking more allocations until the top of the frame is hit. At this point the next largest allocation is mapped immediately to the left of the first, and the process repeats.

Based on testing in [3], this algorithm produces, on average, a normalized value of 0.0422 unallocated slots, and 0.0059 over-allocated slots. The computational complexity in the worst-case scenario is $O((rn)^2)$, where n is the number of allocations and r is the number of enumerated possible rectangular mappings per allocation.

6.2 eOCSA

eOCSA is a variation on OCSA designed to decrease computational complexity. Instead of enumerating every possible rectangular allocation for each allocation, it considers only the minimum-width allocation. For a specific allocation *i* of total size A_i, the width W_i, and Height H_i are calculated using equations $W_i = \lceil A_i / H \rceil$ and $H_i = \lceil A_i / W_i \rceil$ on a subframe of total height H, where $\lceil\ \rceil$ represents a ceiling function. This does not ensure an ideal mapping, but provides a reasonably good approximation and saves computation time enumerating all possible allocations. Because we allocate the minimum possible height for the narrowest mapping of a given burst, we know that the number of over allocated blocks in a mapping will never be higher than that width. This leads to very low over-allocation amounts despite the very efficient method used to choose the sizes. The mappings of allocations above the top of a given column seek to minimize height instead of width, while maintaining a maximum width equal to the initial allocation as before.

According to [2], the average normalized performance relative to the ideal mapping for eOCSA is 0.0614 unallocated slots, and 0.0088 over-allocated slots. Both of these are roughly 1.5 times larger than the equivalent parameters in the OCSA simulation, but the computational complexity of finding this solution is only $O(n^2)$, since enumeration simply does not occur.

6.3 mOCSA

Both OCSA and eOCSA work well when only relatively small packets are added to the regions at the top of each column. Small packets can easily be added to the top of columns of even relatively small widths. However, larger packets may not fit in these narrow regions, even if the total allocation space remaining above all the columns combined could easily fit them.

The goal in these algorithms of allowing the packets at the top of the subframe to only be as wide as the large allocation at it's base is to avoid impacting future allocations by limiting their potential height. Given that these allocations are larger than others, it makes sense to give them priority in allocation shape to limit over-allocation as much as possible. However, this can be accomplished without limiting the allocation in the higher-area to the extent that they do.

Merging OCSA (mOCSA) is a new algorithm designed to allow better use of the higher areas of the subframe. In mOCSA, the entire lower-region is allocated first, and the unallocated region above each column is tracked. Once no further packets can fit along the bottom of the subframe, we analyze these regions, and modify them to increase our allocation options. Because we can only allocate rectangles, and adjacent regions may be different heights, combining them will generate one area spanning the width of both regions, and one that occupies the leftover space. If the wider of these regions is larger than either of the originals, we perform the merge. Otherwise we leave them unchanged.

Once any merges occur, we allocate bursts into these regions in the same way that eOCSA allocates bursts into the regions above individual columns. That is to say given a burst of length l and an allocation space Ai of width W_i and height H_i, we map it to a height $H_a = \lceil A_i / W_i \rceil$ and width $W_a = \lceil A_i / H_a \rceil$. This is the narrowest allocation possible at the minimum possible height. The remainder of the allocation space can

then be used for subsequent mappings if there is room. As bursts are mapped into a region, that region's remaining height is updated appropriately, and a new region composed of any leftover width is created. A more computationally intensive version of this algorithm would reexamine the entire set of allocation regions, merging them as appropriate, after each mapping and resulting reduction in the size of a given region.

Like eOCSA, mOCSA is $O(n^2)$, though the coefficient is higher. As the algorithm is effectively identical save the region-merging process in which each region is compared to each other region. Because the set of regions corresponds to the set of bursts mapped to the bottom of the subframe, there will be at most n regions compared, yielding $O(n^2)$ total comparisons.

6.4 Sample eOCSA and mOCSA Mappings

In this section, we present a sample set of allocations and show how both mOCSA and eOCSA would map this set of bursts. The list of allocation sizes is found in Table 6.1. The mapping produced by eOCSA is found in Figure 6.1, the mapping produced by mOCSA is found in Figure 6.2, and an ideal mapping is found in Figure 6.3.

This example illustrates the problem with the strictly column-based approach used by eOCSA and OCSA. While there is a very large unallocated region in the eOCSA-generated mapping, the algorithm is limited to allocating blocks the same width as the columns at the bottom of the grid. Since none of the unallocated bursts will fit in the portion of these columns remaining, none are mapped. This results in a total of five bursts, numbers 15, 16, 17, 19, and 20 not being mapped to the allocation grid. A total of 70 blocks are unallocated in this map, with no over-allocations.

There are several differences in the mOCSA map. The lower regions are distributed slightly differently, as the higher regions are ignored until the entire width is filled at the

bottom. This means that allocation 5 is not mapped above allocation 2, and instead receives it's own column. The more significant difference is that the large region at the top of the frame that is unused in eOCSA is partially filled. In this mapping, only allocations 18, 19, and 20 are unmapped. This results in 39 unallocated blocks, with 2 over-allocated blocks. This is a clearly more effective map than the one generated by eOCSA.

A more computationally intensive version of mOCSA would re-analyze the set of allocation regions after each burst is mapped, and recombine as appropriate. This would allow the algorithm to generate a rectangle of 4 blocks by 3 blocks directly above the location that allocations 9-12 are mapped. Allocation 20 would fit in this new region. This reanalysis is computationally expensive, so it is not generally performed.

Neither map is actually ideal. As shown in Figure 6.3, it is possible to map all bursts into this subframe. However, no known algorithm will generate an ideal mapping quickly enough to be employed.

Table 6.1 Sample Allocation Sizes

Allocation Number	Allocation Size (Slots)
1	44
2	36
3	28
4	26
5	24
6	22
7	22
8	21
9	20
10	20
11	20
12	20
13	16
14	16
15	15
16	15
17	15
18	14
19	13
20	12
21	1

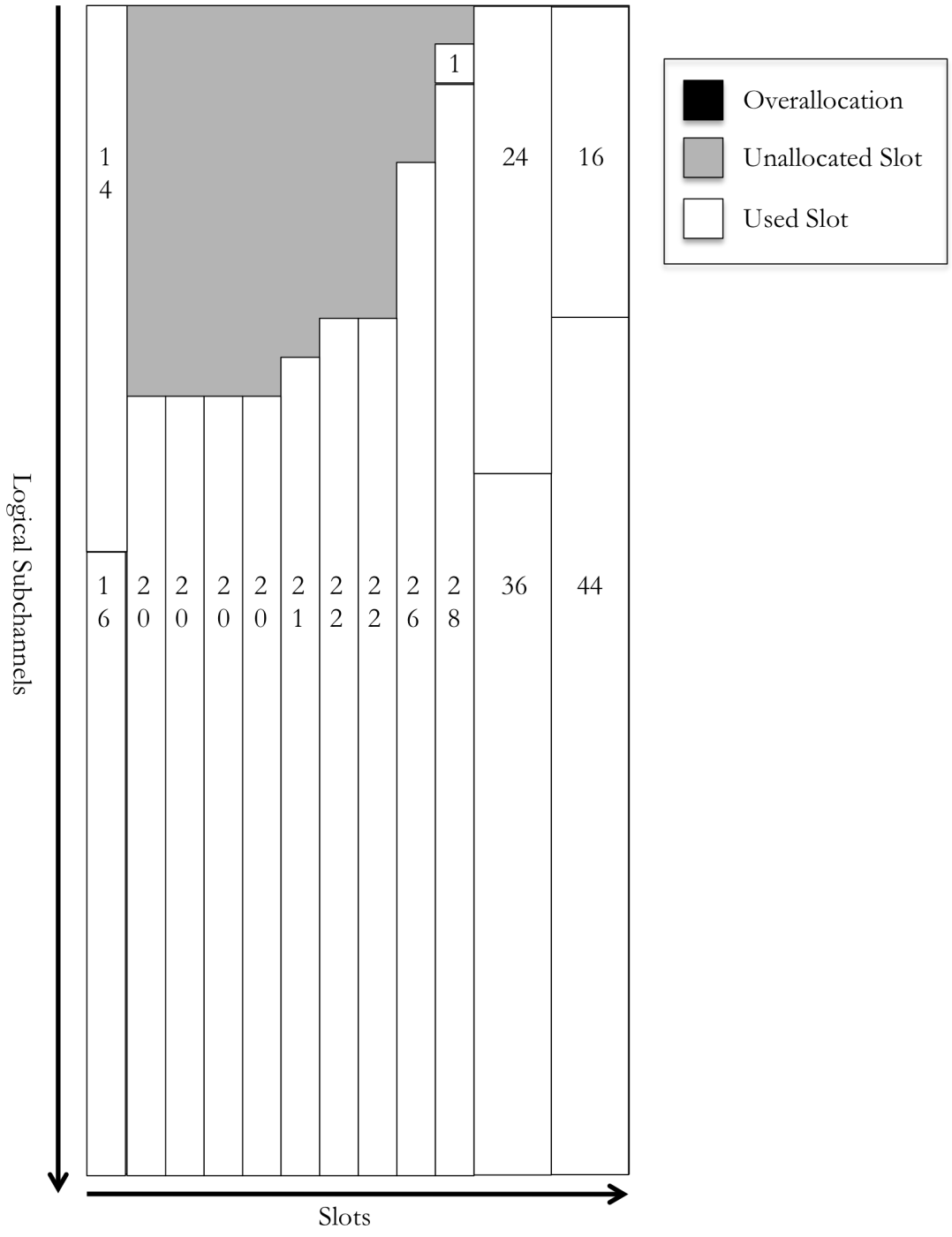

Figure 6.1 Sample eOCSA Mapping

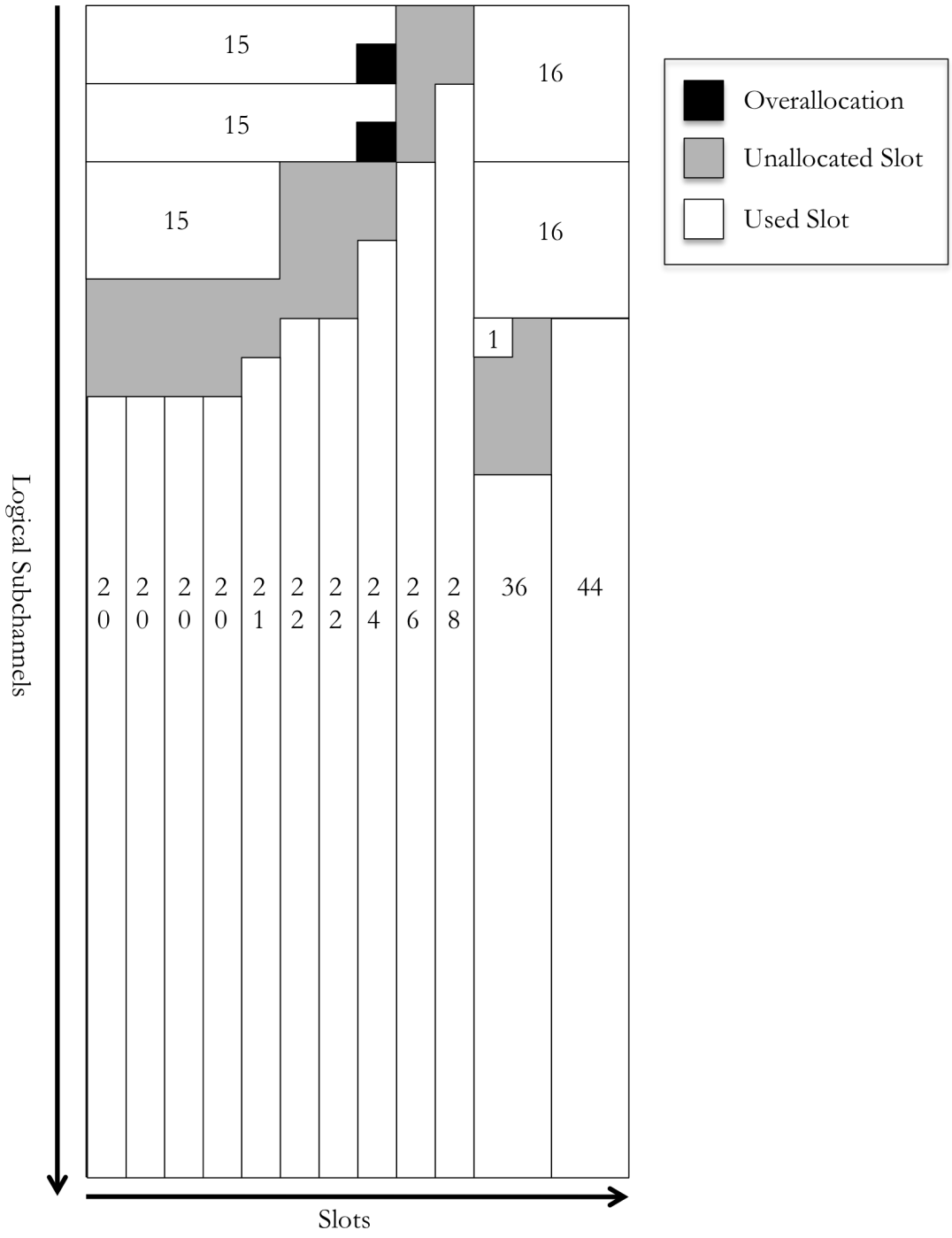

Figure 6.2 Sample mOCSA Mapping

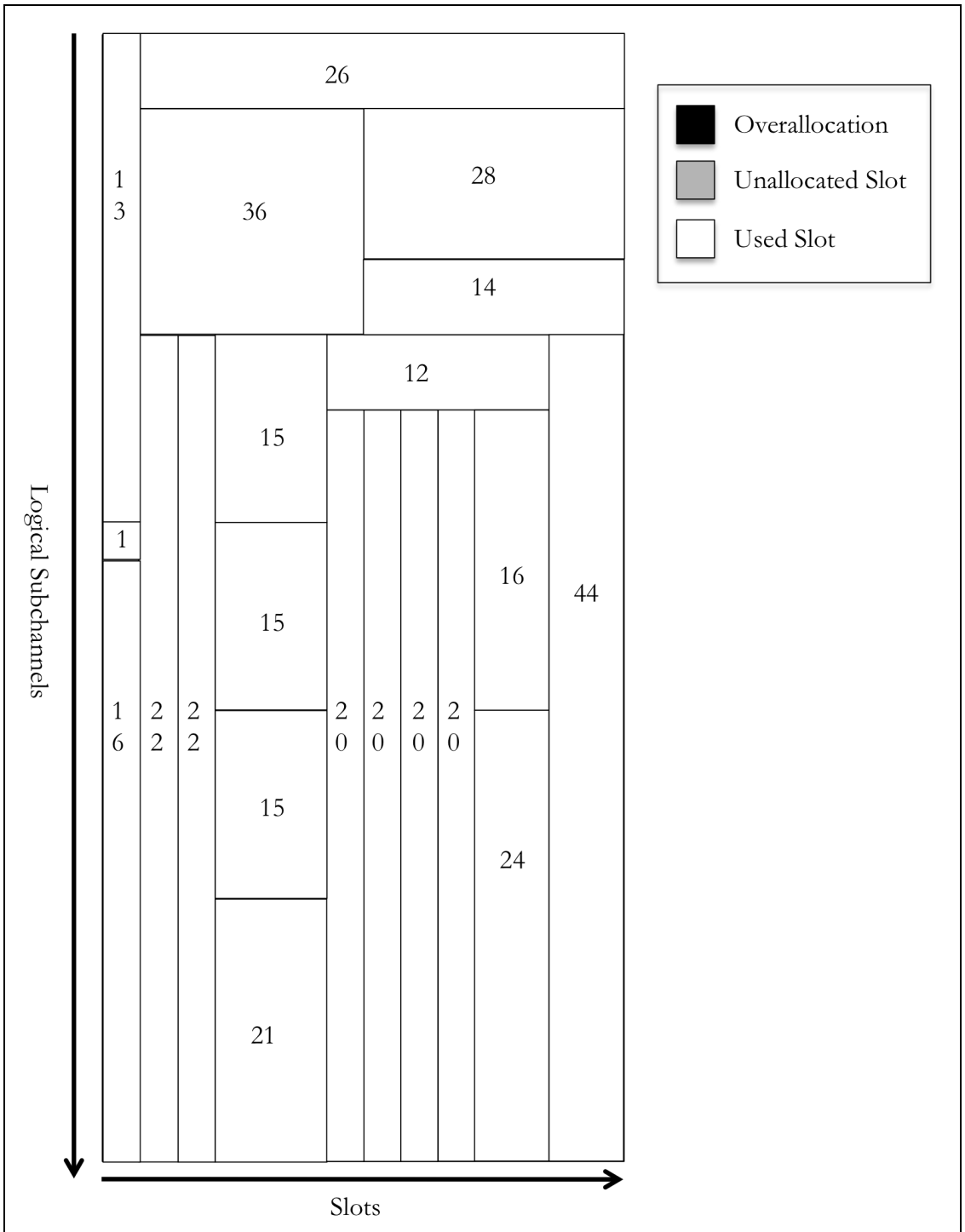

Figure 6.3 Sample Ideal Allocation

6.5 Performance Analysis

In this section we present a comparison of the map quality generated by eOCSA and mOCSA. Data was gathered using the NS-3 model. The simulation script defined a network with a slowly increasing number of subscriber stations, each of which was sent a random amount of data. All service flows used the best effort QoS class, which has no requirements. In the event that all bandwidth requirements could not be filled, a round robin allocation system was employed. As the number of SSs, and thereby the total required throughput increased, the wait for each SS between transmissions also increased. This meant that more data was enqueued to each before it's turn for transmission. Once the network reached it's maximum throughput, the wait times very quickly increased to the point where every SS could fill the entire subframe. This led to single-burst subframes which are obviously trivial to map. Similarly, as traffic was slowly increased from very low levels, early frames had low total allocation requests, resulting in many perfect frames. To avoid either of these skewing the results, a segment of data comprising 2200 was selected in which the total requested allocation was high enough to make mapping non-trivial, but wait times were still low enough to provide frames with many bursts. The simulation was run twice, once with mOCSA and once with eOCSA. Random seeds were held constant so the same set of traffic was provided to both schedulers.

There are several values that can be used to evaluate the quality of a map. By plotting them against the number of bursts that we are trying to map into the allocation grid, we can see how they scale with increasing load on the network for each allocation system. The specific values we are examining are the total number of unmapped bursts (Figure 6.4), the total number of unmapped blocks (Figure 6.5), and the total number of blocks that are wasted. A wasted block is defined as a block that is either allocated to a burst beyond the total size of that burst, or is not allocated to any burst, despite at least one burst not being transmitted in that frame due to lack of space. We ignore unallocated

blocks in frames where every burst is mapped, because there is no potential better use available. Data on the number of wasted blocks is shown in Figure 6.6.

All data points are averages over many frames with the indicated number of requested bursts. The data clearly indicates that mOCSA will usually have fewer unmapped bursts, which will each be somewhat smaller, and fewer blocks will go unused due to either over-allocation or un-allocation.

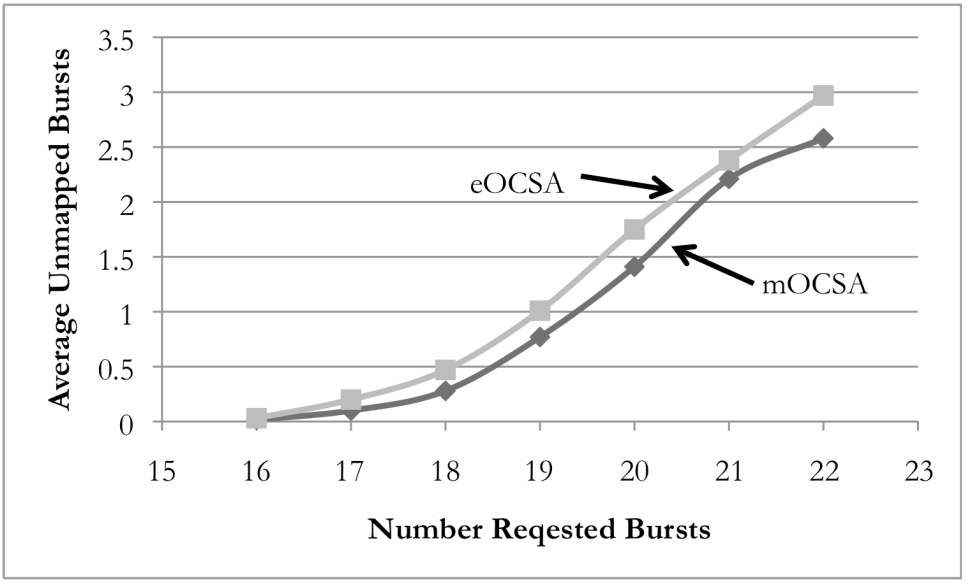

Figure 6.4 Average Unmapped Bursts

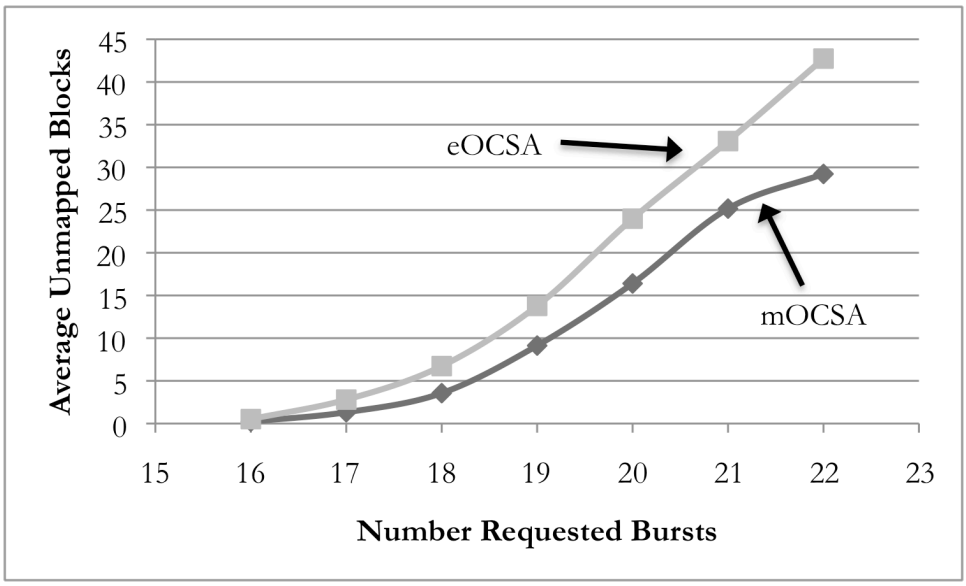

Figure 6.5 Average Unmapped Blocks

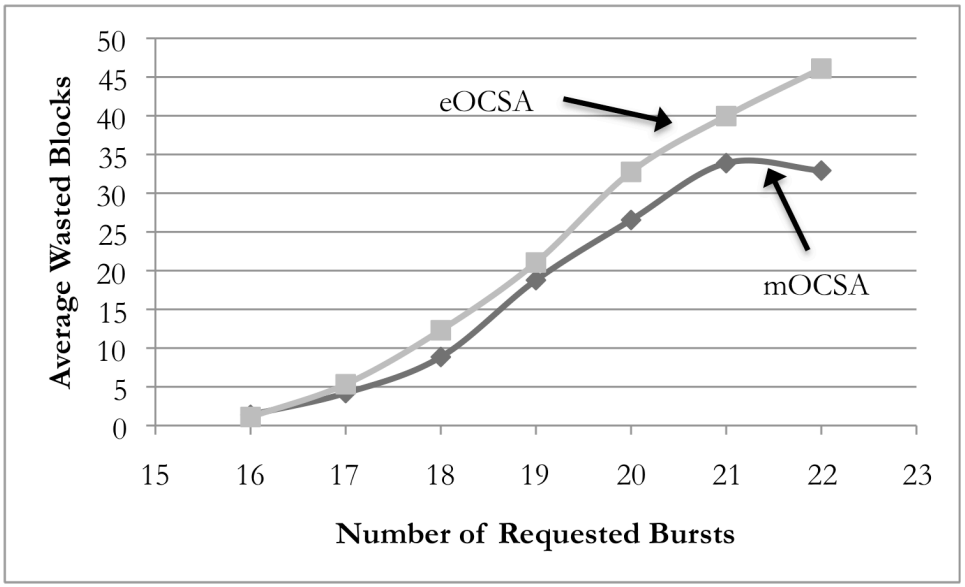

Figure 6.6 Average Wasted Blocks

Chapter 7

Summary

NS-3's architecture is a significant departure from NS-2. This means that translation of NS-2 models to NS-3 requires major overhauls. This thesis discussed the differences between the simulation environments, using the WiMAX Forum's NS-2 model as a case study.

First we discussed the major differences between NS-2 and NS-3. One of the most obvious is the use of different programming language used for scripting. Others include the availability of smart objects, new architectures for Packets and Nodes, and the ability to output industry-standard trace files for analysis in a variety of different tools.

We next provided a basic overview of major components in WiMAX networks, including the OFDMA-based physical layer and the scheduling mechanisms employed in the MAC layer. After providing this overview, we discussed the NS-3 implementation of these mechanisms in the translated model. This included a description of the pre-transmission and post-transmission processing that occurs on packets, and a breakdown of significant sections of the model. These sections include the state machines driving each base station or subscriber station on the network, the system used to classify traffic from IP and MAC addresses to WiMAX Connection IDs, the physical layer, the system used to generate and communicate bandwidth grants and the schedules for both downlink and uplink subframes, various timers and headers that are used, and several remaining classes that did not fit into any of the above categories.

Having established the details of the NS-3 implementation, we discussed the specific changes required from the NS-2 version. This includes the modifications required to achieve the same level of functionality, the enhancements made to provide OFDMA

support, and the issues involved with using NS-3 visualization system. One of the key aspects of the necessary modifications involved NS-3's new packet architecture, and the changes that it required. NS-3 requires that packets be manually packed and unpacked between structures representing the data involved and a stream of bytes that could actually be transmitted, while NS-2 allowed direct access to the headers. The new packet architecture did allow for a much more intuitive method for handling the fragmentation and packing of packets into bandwidth allocations by directly splitting and recombining series of bytes instead of using lists of independent packets.

Finally, we provided a new OFDMA downlink subframe mapping algorithm called merging OCSA, or mOCSA. It is a modification of eOCSA that allows for larger allocations to be mapped into the top-region of the subframe. We then used the NS-3 model to simulate eOCSA and mOCSA, and demonstrated that mOCSA produces consistently better maps in terms of average wasted blocks, average unmapped blocks, and average unmapped bursts.

References

[1] "About the WiMAX Forum," WiMAX Forum, June 2001, http://www.wimaxforum.org/about

[2] Chakchai So-In, Raj Jain, Abdel Karim Al Tamimi, "eOCSA: An Algorithm for Burst Mapping with Strict QoS Requirements in IEEE 802.16e Mobile WiMAX Networks," Proceedings of the Second IFIP Wireless Days Conference, Paris, France, 14-16 December 2009.

[3] Chakchai So-In, Raj Jain, Abdel-Karim Al Tamimi OCSA: An algorithm for Burst Mapping in IEEE 802.16e Mobile WiMAX Networks, Proceedings 15th Asia-Pacific Conference on Communications (APCC 2009), 8th-10th Oct, 2009, Sanghai China.

[4] Gustavo Carneiro, "NS-3: Network Simulator 3." UTM Lab Meeting April 20, 2010. http://www.nsnam.org/tutorials/NS-3-LABMEETING-1.pdf

[5] Joe Kopena, "NS-3 Overview," March 19, 2008. http://www.nsnam.org/docs/ns-3-overview.pdf

[6] Kevin Fall (Ed), Kannan Varadhan (Ed), "The NS-2 Manual," 2010. http://www.isi.edu/nsnam/ns/ns-documentation.html

[7] "The NS-3 Manual," The NS-3 Project, 2010. http://www.nsnam.org/docs/release/3.10/manual/singlehtml/index.html

[8] "The NS-3 Tutorial," The NS-3 Project, 2010. http://www.nsnam.org/docs/release/3.10/tutorial/singlehtml/index.html

[9] Raj Jain (Ed), "WiMAX System Evaluation Methodology, V2.1", WiMAX Forum, July 7, 2008.

[10] Wlia Weingärtner, Hendrik Vom Lehn, Klaus Wehrle, "A Performance Comparison of Recent Network Simulators", Proceedings of the 2009 IEEE International Conference on Communications.

[11] Thomas R. Henderson, Sumit Roy, Sally Floyd, George F. Riley. "ns-3 Project Goals" http://www.nsnam.org/docs/meetings/wns2/wns2-ns3.pdf

[12] IEEE Std 802.16™-2004, IEEE Standard for Local and metropolitan area networks: Part 16: Air Interface for Fixed Broadband Wireless Access Systems.

[13] IEEE Std 802.16e™-2005. Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands

Vita

Christopher Thomas

Date of Birth	June 6, 1987
Place of Birth	Summit, New Jersey
Degrees	B.S. Computer Science, May 2009 M.S. Computer Science, May 2011
Professional And Honor Societies	Association for Computing Machines Upsilon Pi Epsilon

May 2011