2009

A Combined WiMAX/DSRC Approach to Vehicular Networking

Nicholas Doyle
University of Windsor

A Combined WiMAX/DSRC Approach to Vehicular Networking

by

Nicholas Charles Doyle

A Thesis
submitted to the Faculty of Graduate Studies through the Department of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
2009

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances in my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

From wireless Internet access at cafes to the expanding popularity of smart phones, ubiquitous Internet access has generated much public demand and research. Supplying broadband Internet access for in-vehicle applications is a research area still in its infancy. This thesis examines the strengths and weaknesses of WiMAX and DSRC, two protocols that have been central in much of the research surrounding in-vehicle network access. The thesis then proposes a novel system structure that combines both of these technologies and adds a network access layer in order to provide a system structure that offers high bandwidth, bounded latency and robust support for the high levels of mobility experienced by vehicle-based users. This provides the network support for applications such as streaming audio, video and Voice over IP. The thesis also describes a demonstration system that partly implements the proposed system structure.

Acknowledgements

I would like to offer my sincere appreciation to Dr. Kemal Tepe for his flexibility to my constantly changing situation, and for keeping with me all the way through my degree. His encouragement has allowed me to bring this work to a successful completion.

I would like to thank Dr. Roberto Muscedere and Dr. Christie Ezeife for volunteering their time to serve on my thesis committee, and for offering the questions that improved this thesis. I also would like to thank Dr. Rashid Rashidzadeh for agreeing to chair my thesis defence.

The unwavering support of my parents, Debbie and Sheldon, my sister Stephanie and my brother Michael has made this thesis possible. Their belief in me helps me to believe in myself.

Finally, I would like to thank the friends I have made along the way, including Laura Gergely, David Forshner and Amanda McAlorum. They pushed me forward when things were rough and inspired me to take the path less traveled.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AAA	Authentication, Authorization and Accounting
AC	Access Concentrator
B-CID	Basic Connection ID
BS	Base Station
BSID	Base Station ID
CHAP	Challenge Handshake Authentication Protocol
CID	Connection ID
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
EAP	Extensible Authentication Protocol
IAG	Internet Access Gateway
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPCP	Internet Protocol Control Protocol
IPSec	Internet Protocol Security
JIT	Just-In-Time
L2TP	Layer 2 Transport Protocol
L3NP	Layer 3 Network Protocol
LCP	Link Control Protocol
MAN	Municipal Area Network
MT-CID	Managed Tunnel Connection ID
NCP	Network Control Protocol
NS	Network Server
ns-2	Network Simulator, Version 2
OFDM	Orthogonal Frequency Division Multiplexing
PAP	Password Authentication Protocol
PM-CID	Primary Management Connection ID
PPP	Point-to-Point Protocol
RADIUS	Remote Authentication Dial In User Service
RS	Relay Station
SF-CID	Service Flow Connection ID
SID	Session ID
S-OFDMA	Scalable Orthogonal Frequency Division Multiple Access
SS	Subscriber Station
T-CID	Tunnel Connection ID
TCP	Transport Control Protocol
TDD	Time Division Duplexing
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network
WBNP	WiMAX Backend Network Protocol
WiMAX	Worldwide Interoperability for Microwave Access

Chapter 1: Introduction and Background Information

1.1 Introduction

Ubiquitous mobile access to the global Internet has long been a topic of research in computer communications. As the Internet becomes more mature and integrated in everyday activities, there is an increasing demand for broadband Internet access. Laptops using wireless Internet has become a common sight at universities and coffee shops. Devices like the Blackberry and iPhone have been leading the public demand to make these rich network services available on handheld devices. Broadband network access in the vehicle has many potential applications, from traveler information and safety systems to delivery of media content and voice communications [**1**]. The increasing demand for these rich mobile applications will necessitate efficient infrastructure for delivery into mobile platforms such as vehicles. One of the competing standards for wide-area wireless broadband networking in mobile applications is Mobile WiMAX. While it has been designed with these mobile applications in mind, previous research [**2**] has found a flaw causing excessive overhead in a dense network situation with many simultaneously active connections in a single network cell. This is of particular concern in vehicular networks where this situation could easily be seen on major roads and highways.

Another standard that has been the focus of much research in vehicular networking is Dedicated Short Range Communications (DSRC). DSRC is part of the Intelligent Transportation System (ITS) plan in the United States. The U.S. Federal Communications Commission (FCC) dedicated 75 MHz in the 5.9 GHz band for communication between vehicles, and between vehicles and roadside stations. The primary goal of this system is safety applications, allowing vehicles to alert each other of traffic conditions and communication of safety messages from the roadside infrastructure. To encourage wider adoption of the technology, a secondary role for commercial applications is envisioned for tasks such as toll collection and Internet access [**3**].

This thesis intends on developing a system to provide robust Internet access using a combination of WiMAX and DSRC technologies that functions better than either technology on its own. It begins with background information on WiMAX and the multihop amendment currently being developed. The system capacity problem for congested systems is described and modeled for the Mobile WiMAX Orthogonal Frequency Division Multiple Access (OFMDA) frame structure.

DSRC is introduced and the feasibility of node clustering in vehicular applications is discussed. A system is proposed that blends both WiMAX and DSRC to provide mobile vehicle network access, using the best features of both technologies in order to solve the congested system overhead problem that exists with a WiMAX-only network.

1.2 Introduction to WirelessMAN and WiMAX

WiMAX (Worldwide Interoperability for Microwave Access) is a trade name developed by the non-profit WiMAX Forum for the commercialization of the IEEE 802.16 WirelessMAN group of standards. A Metropolitan Area Network (MAN) is defined by the IEEE as being "optimized for a larger geographical area than a LAN, ranging from several blocks of buildings to entire cities". The WiMAX certification program, based on the highly successful Wi-Fi Alliance used to market Wireless LAN, ensures interoperability of equipment.

1.2.1 Mobility in WiMAX

The current revision of WiMAX in use, known as Mobile Wimax [**4**] [**5**], is based on the 802.16e amendment to the IEEE 802.16-2004 release of WirelessWAN [**6**]. Mobile WiMAX has focused on frequencies between 2-11 GHz for non line-of-sight applications and an S-OFDMA frame structure. One important feature about the S-OFDMA system used in Mobile WiMAX is the easy scalability of the system to adopt to varying bandwidth configurations (1.25-20 MHz) in a cell, while keeping other system parameters (frame size, subchannel size, etc.) constant and allowing for simpler hardware. Adaptive modulation and coding allows each individual connection to adjust for changing channel conditions and provide the optimum balance between link throughput and robustness. The previous standard, Fixed WiMAX, used an incompatible Orthogonal Frequency Division Multiplexing (OFDM) frame structure.

1.2.2 Multi-hop WiMAX

The current Mobile WiMAX standard offers only a flat point-to-multipoint network, where all nodes communicate directly with the base station for network access. This limits the network structure, leaving network providers to either increase the power of their base stations or install more base stations to improve system capacity and coverage. The cost of installing and operating WiMAX base stations makes adding extra stations an expensive proposition. One of the ongoing developments for WiMAX is the support for multi hop relaying. A task group is currently working on IEEE 802.16j, the Multi hop Relay Specification for IEEE 802.16, with release scheduled in 2009 [**7**] [**8**]. This amendment proposes the use of relay stations to improve system coverage. With a lower complexity than a base station and a subset of its capabilities, these relay stations reduce the cost of improving system capacity while being transparent to the end user. The terms IEEE 802.16j and 16j will be used interchangeably throughout this thesis.

The major scenarios for using relay station discussed by the working group included fixed infrastructure to improve reception coverage, in-building relaying to improve performance within buildings, temporarily boosting coverage for large events or emergency situations, and coverage on mobile vehicles. The final point is relevant to Internet access in vehicular applications.

While the draft 16j amendment discussed both in-band and out-of-band relaying, in-band relaying has thus far had the most attention. In this scheme, the WiMAX frames are further subdivided to provide multiple transmission zones for the relay stations. This subdivision of bandwidth comes with cost, with less bandwidth available for direct connections to the base station node. Out-of-band relaying could therefore be very attractive, particularly if low transmission power allows for efficient frequency reuse.

1.3 Introduction to DSRC

Another technology for in-vehicle communication that has experienced enormous amounts of research is Dedicated Short Range Communications (DSRC). The physical layer of DSRC is described in the upcoming IEEE 802.11p amendment and is known as Wireless Access in Vehicular Environments (WAVE). The physical layer operates on a dedicated set of frequencies in the 5.9 GHz band and operates as a slightly modified version of the IEEE 802.11a (Wireless LAN) protocol, designed to be more robust for mobile safety applications. The bandwidth is divided into seven 10 MHz channels. Three channels are reserved for safety and public notification usages. The remaining four service channels are available for both safety and non-safety applications [**3**].

Point-to-point routing limits the ability to access the Internet to direct access with roadside units (RSUs). As DSRC has a maximum range in the 100s of metres, this limits the connection time for RSUs, with much of the communication being consumed with association between RSUs and on-board units (OBCs). One improvement that has seen much attention is the overlaying of mobile ad hoc routing protocols onto a DSRC network. This allows a more stable routing system for individual users and Internet access while not in direct range of a roadside station [**9**]. However, recent research [**1**] suggests that DSRC is probably not suitable for broadband Internet access due to the high cost of infrastructure investment.

1.4 Introduction to IP Mobility

The Internet Protocol (IP) was originally developed for relatively stationary networks and nodes. Due to the hierarchical nature of routing tables, particularly in IPv4, IP addresses are tied to network locations and machines that moved had to be assigned a new network address to be reachable. While computers had portability, they didn't have mobility, since the change in network address required re-establishment of all connections.

The first major modification to IP to add support for mobility was Mobile IP [**10**]. This allowed a user to have a permanent universally routable IP address (controlled by the home agent) while using a temporary IP address when it is roaming away from its home network. At the remote location, a control node called the foreign agent registers with the home agent. All traffic destined for the user at its permanent address is intercepted by the home agent and forwarded to the foreign agent, where it is relayed to the user. This process operates transparently to end user applications.

While Mobile IP was designed to handle periodic mobility, the overhead of the system made it unsuitable for systems with high levels of mobility, such as vehicular-based systems. Schemes to address the rapid mobility of nodes within a limited network space have come to be known as micromobility [**11**]. Of the proposed schemes, Cellular IP (CIP) [**12**] is one of the most discussed. CIP divides mobility into two zones – global mobility and local mobility. Local mobility encompasses movement within a single domain, such as within a network provider's network. The local domain is separated from the global Internet with a gateway node. By dividing the network into domains, location updates are optimized for the part of the route that is rapidly changing (the routing between base stations), instead of imposing overhead across the entire global route.

Within the local network, a simple cache-based next-hop routing system is used within the hierarchy of the routers. Two sets of caches are used: routing caches and paging caches. Routing caches are updated with every packet and record the next hop routing information for active users. The paging cache is updated periodically (and after handovers) to store the routing location for idle users. Periodic updating helps keep the overhead of inactive users in check, while still making all nodes addressable.

The user's IP address itself does not have routing significance within a local network using Cellular IP. As a result of this, the user node can retain the same IP address through local mobility. When a user changes the serving host, messages work backwards through the routers, updating the routing tables in intermediate nodes until routing for that user is fully up to date. Figure 1.1 shows an example of CIP routing within a local network. The mobile node enters the system through base station node D. Through the hierarchy of the system, it is routed through nodes C and A back to the gateway node to the network. Each node updates their routing tables as traffic passes through. As a user node changes access points, the routing back to the gateway is updated. As the user's IP address remains the same and routing beyond the gateway node is unaltered, the end user applications are unaware of the mobility.

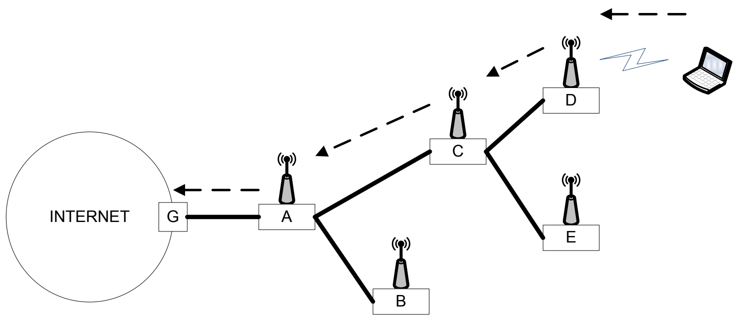

Figure 1.1: Cellular IP Routing Hierarchy

1.5 Introduction to Network Tunneling

Encapsulating tunnelling protocols are a pervasive feature of Internet access systems and computer networks in general. They allow transmission of network data across an incompatible network technology, or allow additional features (such as security) not normally afforded the connection. One of the most commonly used and extended protocols for network access is the Point-to-Point Protocol (PPP). Work on the protocol begun in 1992, culminating in the release of RFC 1661 [**13**] in 1994. With the raise in popularity of the Internet, PPP provided a standardized and complete method for connecting a user to the Internet over a serial link (primarily for dial-up modem connections).

PPP operates over a simple serial link layer to provide a network tunnel from a remote service. The server end provides the necessary configuration information (IP address, etc.) and routes traffic to and from the Internet. PPP is generally used in conjunction with an authentication server. The Remote Authentication Dial In User Service (RADIUS) [**14**] has become the standard for authentication and auditing of PPP connections.

With the advent of broadband, PPP has seen further use in protocols such as PPP over Ethernet (PPPoE) [**15**]. Digital Subscriber Lines (DSL) has become a common method to provide broadband Internet access. The DSL modem provides a raw Ethernet connection into the server provider's access network. While Internet access is possible over a direct Ethernet link layer, it lacks the capability for easy authentication and monitoring of individual users. In response, a version of PPP called PPPoE was developed to provide a PPP-style network tunnel. Rather than functioning over a raw serial link, the point-to-point connection is connected directly over the Ethernet link interface (using MAC addressing). The encapsulation of Internet traffic occurs between the end user PPPoE client and the DSL Access Concentrator (DSL-AC) on the provider side.

The IP Security (IPsec) suite of protocols was developed to secure the Internet protocol, through authentication and encryption of the data stream [**16**]. A public key exchange is used for the initial negotiation and session key generation. Session data is encrypted using a stream cipher, such as 3DES or AES. IPsec has two major modes of operation: transport mode and tunnel mode. The transport mode secures the payload data, leaving the header in plain text. In tunnel mode, the entire IP packet is secured and encapsulated within another packet. This hides routing information beyond the two end-points in the tunnel. When operating in tunnel mode, the IPsec system creates a layer 3 (network layer) tunnel between the two end-points. This is transparent to the end user applications tunnelled across the connection.

Tunnelling protocols are also essential for a Virtual Private Networks (VPN). This is a system that provides a virtual connection to a private network, routed as encapsulated data across the public Internet. This allows organizations to provide access to their internal network to individual users and remote sites across the Internet instead of using high cost dedicated lines, while not compromising the privacy of the network.

One of the most common protocols for VPN systems is the Layer 2 Tunnelling Protocol (L2TP) [**17**]. L2TP is another example of a virtual connection (in this case a layer 2 connection), as it is actually a session (layer 5) layer protocol operating over a UDP connection. A virtual network device gives end user applications the illusion that they are connected directly to the remote network, with the L2TP link operating transparently in the background. A common configuration for VPN access is IPsec encapsulated L2TP packets, tunneling a PPP connection [**18**]. The IPsec encapsulation helps protect an otherwise plain-text communication. The L2TP connection provides a tunneled linked layer over which the PPP protocol can authenticate and open a network (IP) link. Figure 1.2 shows the multiple encapsulations present in communication between the client and the server using this configuration.

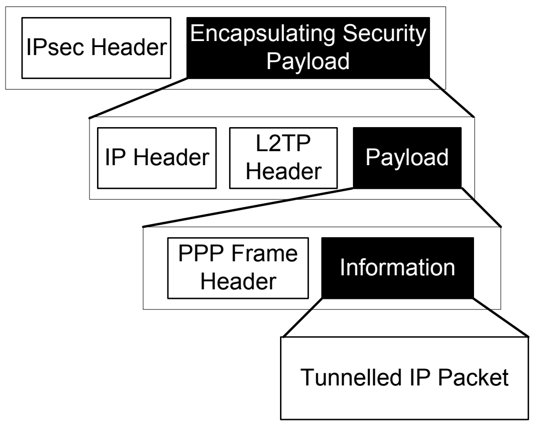

Figure 1.2: Encapsulation in the Layer 2 Transport Protocol

1.6 Thesis Objectives

This thesis details the development of a novel system structure to provide robust Internet access for vehicle-based platforms. This system maximizes the bandwidth available for user applications, controls the latency for messages travelling through the system and minimizes the protocol overhead. The system described in the system is also designed to be robust against the heavy mobility levels inherent in vehicle nodes, both in physical velocity and the rate of migration through the system. The proposed system provides Internet connections that are designed to maintain connectivity through these high rates of mobility. This will support applications such as streaming audio, video and Voice over IP (VoIP).

The unique work presented in this thesis includes a full calculation of system capacity for Mobile WiMAX systems with various levels of system congestion and the use of a combined

WiMAX/DSRC system to concentrate connections and improve system efficiency. This thesis extends the 16j amendment to WiMAX by proposing relaying across a non-WiMAX medium. This thesis proposes a full system structure to achieve the goals of a robust system for vehicle Internet access. This includes the combined WiMAX/DSRC link layer connection and a network layer protocol, based on PPP, which provides the robust Internet connection required for highly mobile vehicle based platforms. The thesis finishes with a demonstration system that was developed to both demonstrate the described system structure and to function as the basis of a future research platform for investigation into vehicular Internet access platforms.

1.7 Thesis Structure

This thesis is organized as follows: Chapter 1 describes the motivation behind the work presented in the thesis, as well as background information into several of the technologies used throughout the thesis. Chapter 2 examines the system capacity issue with congested WiMAX systems and proposes a novel hybrid WiMAX/DSRC structure to address this limitation. Chapter 3 describes in detail the proposed WiMAX/DSRC system structure, including packet structures and the network entry/handover processes. Chapter 4 describes a network layer protocol that operates on top of the WiMAX/DSRC system and provides Internet access for user applications while being robust for the high levels of mobility in vehicle-based systems. Chapter 5 details a demonstration system developed to demonstrate the system structure described in Chapter 3 and 4, as well as forming the core of a future research test system. Chapter 6 concludes the chapter with recommendations for future research work into the proposed system.

1.8 Summary

This chapter provided an introduction to WiMAX, a next generation wide-area networking technology that is a candidate for 4th generation networking systems. The 16j amendment currently under development will allow for more complex network structures by introducing relay nodes. The chapter also discussed DSRC, a technology developed for vehicle-to-vehicle networking that experienced a lot of research. Also introduced in this chapter were the concepts of IP mobility and network tunnelling. These are key components to the system that will be proposed in later chapters in this thesis.

With this background, the next chapter will provide an in-depth analysis of the system capacity of Mobile WiMAX systems. In particular, the overhead produced by the system as the number of active users in the system increases will be calculated. These results will be discussed and possible solutions to improve the efficiency of the system will be introduced.

Chapter 2: System Capacity

2.1 Introduction

This chapter explores the system capacity of a Mobile WiMAX system. Previous research has found that a WiMAX system with many simultaneous connections suffers from a ballooning amount of system overhead. This reduces the amount of bandwidth available for user data transmission. These calculations are re-done for the frame structure used in Mobile WiMAX.

Later in the chapter, solutions to this shortcoming are examined. In particular, clustering of nodes is examined as a potential solution. The potential for efficiency improvement is examined for a variety of system conditions and cluster configurations. This system structure sets the stage for a system description in subsequent chapters.

2.2 Previous Work

Previous research done on WiMAX has suggested that large numbers of simultaneous active connections over a WiMAX link can result in a serious degradation of available bandwidth due to system overhead. This analysis [**2**] found that upwards of 50% of the bandwidth is consumed by overhead in a system with 60 active bi-directional connections.

These findings used other research [**19**] that described a method to calculate overhead in a WiMAX system. The individual overhead components of each frame were described and compared against the total bit capacity of the channel to determine the overhead. Some overhead was fixed, while others varied as a function of the number of active connections in the system. This work analysed the frame structure used in Fixed WiMAX, which is based on an Orthogonal Frequency Division Multiplexing (OFDM) modulation scheme.

2.3 Updated Calculations

The analysis done in previous work used the Fixed WiMAX standard with an OFDM frame structure [**2**]. Mobile capable WiMAX described in the Mobile WiMAX standard uses a somewhat different frame structure in order to preserve the robustness of the connection in highly mobile situations. Rather than OFDM, Mobile WiMAX uses a Scalable Orthogonal Frequency Division Multiple Access (S-OFDMA) frame structure [**4**]. Rather than dedicate all subcarriers to a single user, as is the case in OFDM, S-OFDMA instead defines bandwidth allocations both in terms of time and in terms of a subset of the available subcarriers.

Figure 2.1 shows the structure of a Mobile WiMAX frame and illustrates some of the sources of overhead. System overhead is produced by fields in the frame not being used to transmit user data. The most important are the DL_MAP and UL_MAP fields, specifying the user bandwidth allocations for the frame in the downlink and uplink directions. As the number of active connections increase, so does the number of entries in each of these fields. Other system components, including the preamble and the ranging slot (in the uplink subframe), further reduces the system efficiency.

Further system overhead comes from unused allocations. It is conceivable that not all of a user's allocated bandwidth will be used. This may be due to a user not having sufficient data to transmit, or when the user's data does not fill the entire allocation. This inefficiency adds up over multiple connections. A conservative estimate would be to assume half of the last allocated slot is empty and therefore contributing to the overhead. If the traffic is burstier in nature (such as VoIP), the losses due to this unused allocation could be much greater.

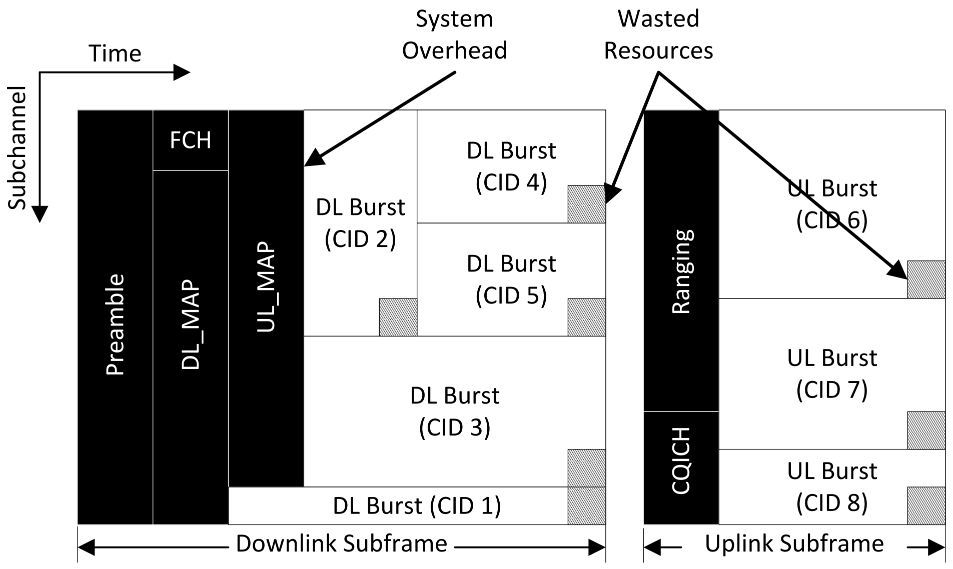

Figure 2.1: Mobile WiMAX Frame

The overall efficiency of the system is defined by the ratio of MAC data transmitted to the total data transmitted. From [**19**], this is given as:

$$\eta = \frac{\theta_{Net/MAC}}{\theta_{Phy/symbol}} \tag{1}$$

The derivations of $\theta_{Phy/symbol}$ and $\theta_{Net/MAC}$ were updated from the original work to reflect the OFMDA system used in Mobile WiMAX. $\theta_{Phy/symbol}$ is the physical bit-rate across the physical interface and is defined as:

$$\theta_{Phy/symbol} = \frac{\left[\left(N_{DL\ Symbols}\right) \cdot \left(N_{DL\ SD} \cdot R_C \cdot \log_2 M\right)\right] + \left[\left(N_{UL\ Symbols}\right) \cdot \left(N_{UL\ SD} \cdot R_C \cdot \log_2 M\right)\right]}{\left(N_{DL\ Symbols} + N_{UL\ Symbols}\right) T_{symbol}} \tag{2}$$

The total number of symbols in the uplink and downlink direction, $N_{DL\ Symbols}$ and $N_{UL\ Symbols}$, are determined by the ratio between downlink and uplink slots selected. $N_{DL\ SD}$ and $N_{UL\ SD}$ are the number of data subcarriers in the downlink and uplink directions. $\theta_{Net/MAC}$ is the net data rate of user data being transmitted and is defined as:

$$\theta_{Net/MAC} = \left[\frac{\left(N_{Payload\ DL} + N_{Payload\ UL}\right) \cdot BpS_m}{T_{Frame}}\right] \tag{3}$$

$N_{Payload\ DL}$ and $N_{Payload\ UL}$ are the total slots available for data transmission in the uplink and downlink directions, with the overhead (as described above) subtracted from the total number of available slots. When multiplied by the bit rate for the selected modulation scheme and divided by the total frame time, the bit rate is found. BpS_m is the uncoded subchannel block size, representing the total amount of user data that can be encoded into each slot. This will vary with the modulation scheme and coding rate used.

2.3.1 Detailed Calculations

The previous section described briefly the method used to calculate the system overhead present in a Mobile WiMAX system. In this section, the method is expanded and described in great detail. The analysis method used is based on that described in previous work [**19**], but had to be substantially modified to accommodate the differences in frame structure.

In Scalable Orthogonal Frequency Division Multiple Access (S-OFDMA), the available bandwidth is broken into a series of orthogonal (non-interfering) subcarrier frequencies, called subcarriers. Some of the subcarriers are dedicated to data transmission, while others are used for tasks such as pilots and guard bands. Groups of subcarriers are put together to form a subchannel, the smallest bandwidth allocation for a user.

The number of available subchannels (and thus users) is limited by the available channel bandwidth. Scalable OFDMA allows for flexible allocation of bandwidth depending on licensed bandwidth and allocation schemes. The four main bandwidth sizes described in IEEE 802.16e are 1.25 MHz, 5 MHz, 10 MHz and 20 MHz. Scalable OFDMA allows for a variety of bandwidth configurations, while all other system parameters (frame length, symbol length, etc.) remain the same. This was a design decision made to simplify the hardware in mobile devices, which weren't anticipated to be very powerful. Mobile WiMAX uses Time Division Duplexing (TDD), dividing the frame into downlink and uplink subframes.

The smallest allocation that can be made in the OFDMA scheme is called a slot, which defines the allocation in bandwidth and time. In the downlink direction, it consists of one subchannel and two symbols in time. In the uplink direction, it is one subchannel and three symbols in time. Table 2.1 shows the number of data carriers available for each bandwidth configuration.

Table 2.1: WiMAX Subchannels by Available Bandwidth

Channel Bandwidth	**FFT Size**	**Subchannels (Down/Up)**	**N_{SD} – Data carriers per symbol (Down/Up)**
1.25 MHz	128 bits	3/4	72/64
5 MHz	512 bits	15/17	360/272
10 MHz	1024 bits	30/35	720/560
20 MHz	2048 bits	60/70	1440/1120

The number of subchannels in the uplink and downlink directions is defined in the IEEE 802.16e standard (8.4.6.1.2.2 for downlink, 8.4.6.2 for the uplink). A subchannel consists of a number of data carriers, which are a set of frequencies that can send encoded user data. Calculating the number of data carriers (N_{SD}) in the downlink and uplink directions (and thus the data throughput capacity) is done as follows:

Calculating N_{SD} for the downlink subframe:

The subchannels for downlink communications are defined by structures known as clusters. A cluster consists of 14 subcarriers spread over two symbols. Over the two symbols, this represents 24 data symbols and 4 pilot symbols. A subchannel consists of two clusters, for a total of 48 data carriers over 2 symbol times. A figure of a cluster can be seen in Figure 2.2.

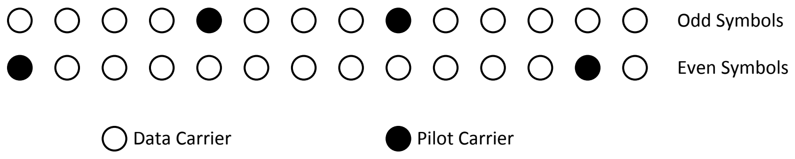

Figure 2.2: OFDMA Cluster Structure (Downlink)

Using a 10 MHz channel as an example, the number of data carriers per symbol (N_{SD}) for the downlink subframe can be calculated to be:

$$\frac{30_{subchannels/symbol} \cdot 24_{data\ carriers/cluster} \cdot 2_{clusters/subchannel}}{2_{symbols/cluster}} = 720_{data\ carriers/symbol} \qquad (4)$$

Calculating N_{SD} for the uplink subframe:

In recognition of the less robust nature of a mobile client, the uplink communications uses a different allocation structure. As shown in Figure 2.3, the available carriers are broken into tiles made up of four subcarriers and three symbol times. More of the carriers are dedicated to pilot signals for increased robustness. A subchannel consists of six of these tiles, for a total of 48 data carriers over 3 symbol times.

This can consist of six consecutive tiles, or a more dispersed allocation for robustness against subcarrier specific interference. Furthermore, the allocated tiles can be changed between frames for more robustness.

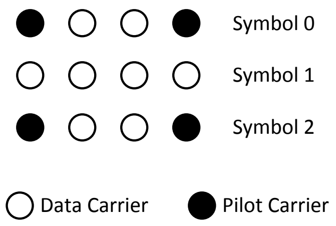

Figure 2.3: OFDMA Tile Structure (Uplink)

Using a 10 MHz channel as an example, the number of data carriers per symbol (N_{SD}) for the uplink subframe can be calculated to be:

$$\frac{35_{\,subchannels} \cdot 6_{tiles/subchannel} \cdot 8_{data\ carriers/title}}{3_{symbols/tile}} = 560_{data\ carriers/symbol} \tag{5}$$

According to the IEEE 802.16e and Mobile WiMAX specifications for the OFDMA modulation scheme, the following modulation/coding rates are defined. Table 2.2 is adapted from the one in [**19**], but updated to the different block size used in OFDMA.

Table 2.2: Coded and Uncoded Block Size by Modulation

PHY mode (m)	**Modulation and coding rate**	**Coding Rate R_c**	**$\log_2 M$**	**Uncoded subchannel block size [bits] (BpS_m)**	**Coded subchannel block size [bits]**
1	QPSK 1/2	1/2	2	48	96
2	QPSK 3/4	3/4	2	72	96
3	16-QAM 1/2	1/2	4	96	192
4	16-QAM 3/4	3/4	4	144	192
5	64-QAM 2/3	2/3	6	192	288
6	64-QAM 3/4	3/4	6	216	288
7	64-QAM 5/6	5/6	6	240	288

The selection of modulation and coding rate is a factor of the channel condition, with more complex modulation and lower coding rates improving the subchannel throughput, but decreasing the resistance against noise and interference. The modulation and coding rate will be dynamically adjusted to provide the best throughput for the channel condition, aided by the fast feedback mechanism defined in Mobile WiMAX.

The physical mode (m) is an indication of which combination of modulation and coding rate was selected. The coding rate indicates the number of bits which will contain redundant information for channel robustness. $\log_2 M$ is the number of bits encoded per subcarrier (where M is 4 for QPSK, 16 for 16-QAM and 64 for 64-QAM).

The term block refers to a group of 48 data carriers, which is the basic unit of data transmission over either two (for downlink) or three (for uplink) symbols. The coded subchannel block size is the number of bits that will be encoded in a block. The uncoded subchannel block size (BpS_m) is the number of user data bits that will be encoded in the block and is defined as:

$$BpS_m = 48 \cdot R_C \cdot \log_2 M \tag{6}$$

48 is the number of data carriers per subchannel. R_c is the coding rate and $\log_2 M$ is the number of bits encoded per subcarrier.

As indicated in the previous section, the overall efficiency of the system can be defined by the ratio of MAC data transmitted over the air interface to the total data transmitted:

$$\eta = \frac{\theta_{Net/MAC}}{\theta_{Phy/symbol}} \tag{7}$$

Where:

$$\theta_{Net/MAC} = \frac{\sum Payload}{T_{frame}} \tag{8}$$

And:

$$\theta_{Phy/symbol} = \frac{N_{SD} \cdot R_C \cdot \log_2 M}{T_{symbol}} \tag{9}$$

The payload is the total number of data bits transmitted and T_{frame} is the frame time (5 ms for Mobile WiMAX). N_{SD} is equal to the number of data subcarriers, R_C is the coding rate, $\log_2 M$is the number of bits encoded by the modulation scheme and T_{symbol} is the symbol time (102.9 µs for Mobile WiMAX including guard band).

n_c is defined to be the number of bidirectional connections active in the system. While connections are allocated on a directional basis, a bidirectional model makes sense for the kind of traffic being investigated in this paper.

Mobile WiMAX incorporates different effective symbol times and number of data carriers for the downlink and uplink subframes, as well as a variable ratio of symbol allocation. Therefore, the physical transmission rate can be further defined as:

$$\theta_{Phy/symbol} = \frac{\left[\left(N_{DL\,Symbols}\right)\cdot\left(N_{DL\,SD}\cdot R_C\cdot\log_2 M\right)\right]+\left[\left(N_{UL\,Symbols}\right)\cdot\left(N_{UL\,SD}\cdot R_C\cdot\log_2 M\right)\right]}{\left(N_{DL\,Symbols}+N_{UL\,Symbols}\right)T_{symbol}} \tag{10}$$

$N_{DL\,Symbols}$ and $N_{UL\,Symbols}$ are the total symbols dedicated to the downlink and uplink and $N_{DL\,SD}$ and $N_{UL\,SD}$ are the number of data subcarriers in the downlink and uplink, respectively. This calculation does not take into account the overhead of preamble, as well as the guarding between the transmit and receive portions of the frame, as they don't contribute to MAC level system overhead.

To calculate the payload, total slots available for data transmission are calculated, with the slots taken up by overhead subtracted. This is calculated to be:

$$N_{Slots\,DL} = \left[\left(\left(N_{DL\,Symbols}-1\right)/2\right)\cdot N_{DL\,SC}\right] \tag{11}$$

And:

$$N_{Slots\,UL} = \left[\left(N_{UL\,Symbols}/3\right)\cdot N_{UL\,SC}\right] \tag{12}$$

$N_{DL\,SC}$ and $N_{UL\,SC}$ are the number of subchannels in the downlink and the uplink, respectively. One symbol is removed from the downlink subframe to account for the preamble.

The number of slots available for payload (user data) is defined as:

$$N_{Payload} = N_{Slots} - \sum Overhead \tag{13}$$

This can be further defined for the downlink and uplink subframes through the specification of the system overhead components specific to each direction as:

$$N_{Payload\ DL} = N_{Slots\ DL} - N_{FCH} - N_{DL-MAP} - N_{UL-MAP} - N_{DCD} - N_{UCD} \tag{14}$$

And:

$$N_{Payload\ UL} = N_{Slots\ UL} - N_{Ranging} - N_{ACK} - N_{CQICH} - N_{Padding}. \tag{15}$$

The system overhead takes the form of the preamble, the Frame Control Header (FCH), the DL-MAP, the UL-MAP on the downlink, and contention channels for ranging, bandwidth request, and fast feedback on the uplink. The preamble is one symbol long and allows for synchronization of all clients.

The FCH contains channel configuration information, including MAP message length, sub-channels, coding scheme, etc. The DL-MAP and UL-MAP messages provide the per-user bandwidth allocations in the downlink and uplink directions, respectively. The FCH consumes 6 slots. The DL-MAP and UL-MAP are dependent on the number of connections. Each consists of a standard MAC header and ends with a CRC, for a total of 10 bytes. Furthermore, each message consists of the standard DL-MAP and UL-MAP message, with an information element (IE) for each connection. The length of the DL-MAP and UL-MAP messages, in bytes, is defined as:

$$L_{DL-MAP} = L_{Header+CRC} + L_{DL-Map\ Header} + L_{PHY\ Sync} + L_{DL-MAP-IE} + L_{Padding} \tag{16}$$

$$= 80 + 72 + 32 + n_c \cdot 60 + 4 = 188 + n_c \cdot 60$$

And:

$$L_{UL-MAP} = L_{Header+CRC} + L_{UL-Map\ Header} + L_{UL-MAP-IE} + L_{Padding} \tag{17}$$

$$= 80 + 64 + n_c \cdot 48 = 144 + n_c \cdot 48$$

Where:

$$N_{DL-MAP} = \frac{L_{DL-MAP}}{BpS_1} \text{ and } N_{UL-MAP} = \frac{L_{UL-MAP}}{BpS_1} \tag{18}$$

The upload and download maps are modulated using 1/2 QPSK, the most robust modulation available.

The DCD and UCD messages define the uplink and downlink channel information, respectively. To conserve bandwidth, these messages are not transmitted every frame. They are instead transmitted periodically. These messages were not included in the calculation as they are a function of implementation (how often they are transmitted). They would function to slightly increase the baseline system overhead in the system.

These calculations assume the system is at a steady state, with a constant bandwidth allocated (a CDMA ALLOCATION IE variant of the IE would be used to change the allocation for the UL). The variant of the DL-MAP-IE with listed CIDs is used (with one CID allocated per IE).

The Mobile WiMAX specification specifies a 6 slot channel for ranging and bandwidth requests. A 6 slot channel is used for channel quality information (CQI) requests, part of the fast feedback mechanism used for mobile systems to adapt modulation based on changing channel conditions. So:

$$N_{Ranging} = 6 \text{ and } N_{CQICH} = 6 \tag{19}$$

Finally, overhead is introduced by each connection, as the MAC PDU will not perfectly align to the MAC SDU and padding will be added. It is reasonable to assume that this adds up, over time, to half of a slot in each direction per bidirectional connection. This is making the conservative estimation that nodes have information to transmit all the time. If the transmissions are burstier in nature, much more bandwidth will be lost in the form of unused allocations.

Using the conservative estimate, the overhead due to this unused bandwidth is found to be:

$$N_{Padding} = n_c \cdot 0.5 \tag{20}$$

With this information, the MAC data bitrate in the downlink and uplink directions is calculated to be:

$$N_{Payload\ DL} = N_{Slots\ DL} - \left[\left(\frac{6+188+n_c \cdot 60+144+n_c \cdot 48}{BpS_1}\right) + n_c \cdot 0.5\right] \tag{21}$$

$$= N_{Slots\ DL} - \left[\left(\frac{388 + n_c \cdot 108}{BpS_1}\right) + n_c \cdot 0.5\right]$$

And:

$$N_{Payload\ UL} = N_{Slots\ UL} - (6 + 6 + n_c \cdot 0.5) \tag{22}$$

$$= N_{Slots\ UL} - (12 + n_c \cdot 0.5)$$

Finally, the MAC data transmission rate is calculated to be:

$$\theta_{Net/MAC} = \left[\frac{(N_{Payload\ DL} + N_{Payload\ UL}) \cdot BpS_m}{T_{Frame}}\right]. \tag{23}$$

2.4 Initial Results

The formulas derived above formed the basis of a MATLAB model of the system, which is included in Appendix A.1. They allowed testing of different configurations of the system, including the number of active connections, the ratio of uplink symbols to downlink symbols and the modulation and coding used for the data being sent. The results, seen in Figure 2.4, shows a decrease in the system efficiency, from 92% with a single active bi-directional connection to 58-59% with 70 active bi-directional connections. The subchannel ratio was also adjusted, using the WiMAX specification downlink:uplink ratio limits of 35:12 and 26:21 and a median value of 30:17. As more resources were dedicated to the uplink, a slight increase in the amount of overhead was seen. These results found an overhead problem with the OFDMA system in Mobile WiMAX which was similar to that found with Fixed WiMAX [**2**].

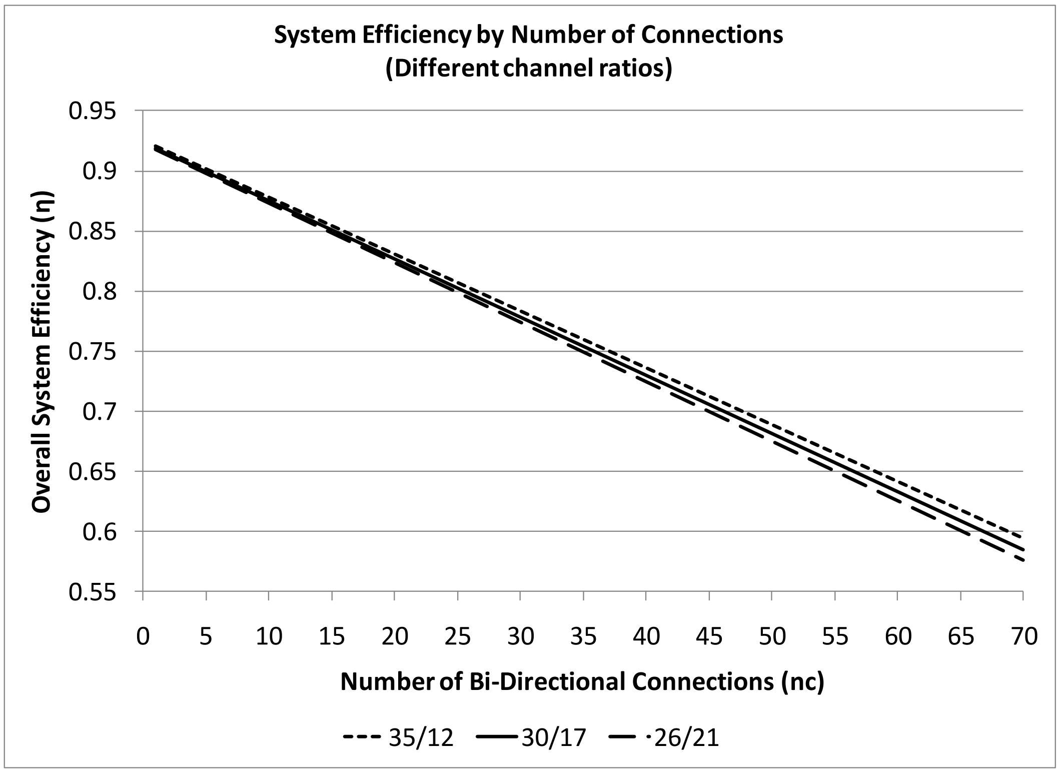

Figure 2.4: System Efficiency by Number of Connections

2.5 Node Clustering

The results of the calculation showed that system efficiency suffered when the systems were congested. This inefficiency stemmed from the bandwidth allocation information included within each frame, coupled by unused parts of allocated bandwidth due to nodes not being able to fill their allocated bandwidth, and is multiplied by number of link allocations within the system. By reducing the number of individual connections within the system, more of the bandwidth can be used for actual data transmission.

This section discusses methods to approach this issue. The concept of concentration of connections, introduced by the 16j amendment, is discussed as a possible solution. Network clustering and vehicular ad-hoc networks are then investigated. Finally, the two techniques are combined and proposed as a possible solution to the issue of system overhead in WiMAX systems.

2.5.1 Concentration and WiMAX

Concentration of connections is one of the items addressed by the IEEE 802.16j amendment. As mentioned in previous research [**2**] [**8**], the 16j amendment introduces the relay station node. This node has some of the functionality of a base station and provides access to the WiMAX network for a number of users. While operating under the control of a single base station node, this node appears to end users as an independent base station. A new class of connection is defined between the relay station nodes and the base station called a tunnel. These connections allow concentration of the connections for all subscriber station nodes being serviced by the relay station, transmitting this traffic across a single connection. This reduces the number of active connections in the system, improving system efficiency while servicing the same number of users.

All of the literature on 16j reviewed for this thesis has thus far concentrated on in-band relaying. As discussed in [**8**], the WiMAX subframes are further subdivided to provide the bandwidth for communications between the relay nodes and the subscriber nodes served by them. However, this is still subdividing a scarce resource (the WiMAX bandwidth). Furthermore, as the relay stations are using the same frequencies one another, interference starts becoming a problem (particularly if the relay stations are moving and close to each other). Furthermore, particularly if non-transparent relaying is used, the relay stations produce an additional set of system header messages. The WiMAX link can be identified as the scarcest bandwidth link in the system, and the one that must be optimized as much as possible.

Although not included in literature on multi-hop WiMAX thus far, the 16j standard also offers support for out-of-band relaying – that is, relaying using a separate set of frequencies. This will be the basis of the proposed network structure. By using a separate set of frequencies for relaying, the full bandwidth of the cell remains unfragmented.

This thesis intends on taking things even further. Rather than use WiMAX for short range relaying, another technology is going to be used. This will be better suited for short-range communications, allowing for better frequency reuse. This is a technology designed (and tested) specifically for vehicle-to-vehicle communications. As will be explained below, this also allows for ad-hoc clustering, which makes sense for mobile vehicle networking.

It should be noted at this point that only a two-hop network is being investigated. That is, there is only a single relay station between the subscriber station node and the base station. The 16j amendment supports (in theory) an unlimited chain of relay stations. For the purposes of this investigation, this additional network complexity is not required (or desired).

2.5.2 Clustering and DSRC

Clustering, simply, is the formation of a tiered structure within a (typically) wireless network rather than using a flat routing structure for communications. In the context of self-arranging ad-hoc networks, clusters are formed by grouping together geographically close nodes. One node with direct communications with all nodes in the cluster is designated the cluster head. All communications between the nodes in the cluster and traffic to and from the cluster goes through this cluster head node.

DSRC is a popular ad-hoc protocol specifically designed for vehicular networking and is based on the popular IEEE 802.11 Wireless LAN (WLAN) standards. It is well suited for short range communication, having a maximum operational range in the 100s of metres. The IEEE 802.16p (WAVE) amendment makes WLAN suitable for vehicle communications [**3**]. As mentioned earlier, DSRC reserves a set of four general purpose communications channels. These characteristics make it well suited for forming clusters of vehicle nodes.

Previous research into using ad-hoc clustering with DSRC [**20**] found that vehicles travelling on highways or major roadways form what is termed a “pseudolinear network”. Constrained by the roadway and relative velocities, nodes travelling on highways and major roads will remain relatively stable with one another, particularly if the road is congested. As such, a linear communication line will form along the roadway. The relative stability over time between the nodes makes these systems ideal candidates for forming cluster groups. The linear structure also makes frequency planning easier in an effort to minimize inter-cluster interference.

In summation, DSRC is a technology for vehicle-to-vehicle communications that has seen much research. By applying the concepts of clustering to DSRC, which has been found in research to be realistic in heavily congested situations, the clusterhead is in a position to concentrate the signals from all the cluster members.

2.5.3 Combined WiMAX/DSRC Approach

The novel approach proposed by this thesis is to use ad-hoc vehicular networking based on DSRC technology to form clusters of vehicular nodes, with the cluster head functioning as an IEEE 802.16j relay node in order to concentrate connections across the wide-area WiMAX link to the base station. The highly congested system conditions where WiMAX overhead becomes an issue is also the situation where clustering of DSRC nodes becomes most effective. Figure 2.5 shows the suggested system layout. The smaller circles represent the short-range DSRC clusters, with one node identified as the cluster heads. The different patterns represent the different clusters, using different frequencies. The identified cluster head is then used to communicate over WiMAX to the base station. WiMAX is used to cover a relatively large geographic area with network coverage.

One of the major problems with a DSRC-only system is the latency issues inherent with ad-hoc routing systems. As the number of hops between the user and a network gateway is dynamic and the links can be fragile, it is difficult to provide bounds on the latency seen by the system. This can be a problem with Internet applications that are sensitive to latency, including voice communications and streaming media. Research [**1**] has also found that using road-side stations for Internet access is impractical without a heavy investment in infrastructure. A combined system also offers the ability to revert to WiMAX-only service, eliminating the sparse node problems seen in an ad hoc network where nodes aren't dense enough to allow for continuous routing. This flexibility allows for rapid adaptation to changing system conditions, such as exiting a busy highway onto a non-busy side road.

While the coexistence of WiMAX and DSRC in a heterogeneous environment has been discussed in literature [**21**], this is the first proposal to merge the two systems to extend the WiMAX protocol across DSRC and use it to provide Internet access in vehicular applications. There are many potential benefits to using this combined system rather than either WiMAX or DSRC on its own. The improvement to the WiMAX system capacity minimizes the installation of costly WiMAX base stations. The use of the separate set of DSRC frequencies negates the need to divide the scarce WiMAX bandwidth for relaying. As DSRC is a short range system, the frequency reuse can be quite efficient.

In the proposed system, all subscriber nodes connected to the WiMAX network through the DSRC cluster head are full members of the WiMAX network. In other words, the cluster head is not simply acting as an Internet access proxy. In this way, all subscriber station nodes are registered onto the WiMAX network and are individually addressed by the system. By doing this, all nodes are better authenticated and audited by the system. This system is also more robust mobility in the system, taking advantage of WiMAX's built-in handover support. Finally, by registering all users on the WiMAX network, it further eases the switch between a direct WiMAX connection and a combined WiMAX/DSRC connection without resetting active connections, and allows for easier handovers between relay station nodes. The switch between a direct WiMAX connection and a WiMAX/DSRC connection does not break the session that the user has with the base station.

Finally, this system will help encourage the adoption of DSRC, which is desired for its safety and ITS applications. The combined network also offers ITS systems an alternative to expensive [**1**] DSRC roadside equipment for data collection, electing instead to use the combined DSRC/WiMAX network to report ITS information such as traffic conditions.

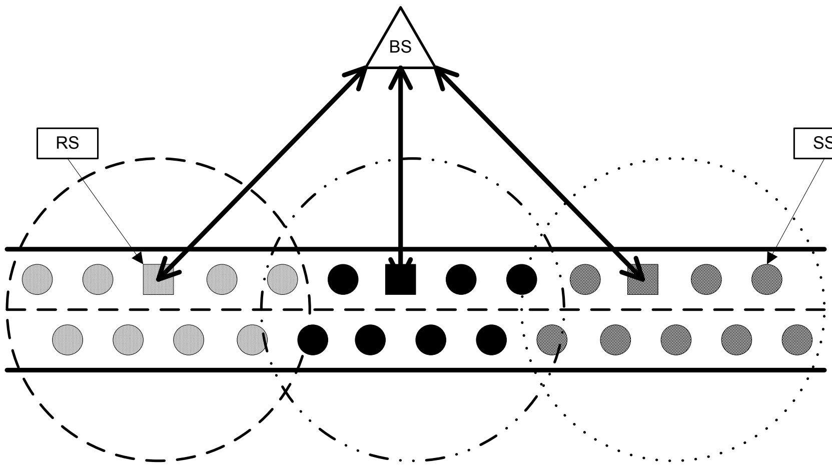

Figure 2.5: Proposed DSRC/WiMAX System Structure

2.6 Results with Clustering

To evaluate the benefits of clustering, the calculations derived in the previous section were repeated using different cluster sizes. The MATLAB code is included in Appendix A.2. Figure 2.6 shows the effect of clustering on system efficiency with various levels of loading. All systems showed an improvement with an introduction of clustering, with heavily loaded systems (ones with higher numbers of active nodes) showing the greatest improvement. Figure 2.7 shows the percentage improvement in efficiency with clustering for these systems. The greatest improvement is seen with introducing clusters of 2- to 4-nodes per cluster, which also coincides with the most drastic reduction in the number of connections. With 40 users in a system, for example, 2-node clusters reduce the total number of connections to 20 and 5-node clusters will further reduce that to 8. This results in an efficiency improvement of 9.45% and 15.13%, respectively. Further clustering will cause further gains, but with a diminishing rate of return. In an extreme case, where a system is loaded with 70 users, 10-node clusters will result in a 30% improvement in system efficiency. As the total number of users in the system increases, clustering will significantly improve the overall system efficiency, especially in heavily loaded systems.

The result of an improvement of system efficiency through clustering is an increase in the available per-user bandwidth in the system. For example, a system with 40 users and no clustering has 534 kBits/s of throughput available for each user. This is found by calculating the $\theta_{Net/MAC}$ value for the number of active connections across the WiMAX interface and dividing by the total number of users in the system. Implementing 2-node clusters (for a total of 20 clusters) increases the per-user throughput to 602 kBits/s, with 5-node clusters (for a total of 8 clusters) further increasing this to 643 kBits/s, an improvement of over 100 kBits/s or 20%. Figure 2.8 shows the per-user throughput in the system for various levels of system loading. Adding 2-node and 5-node clusters shows a constant improvement over the non-clustered system.

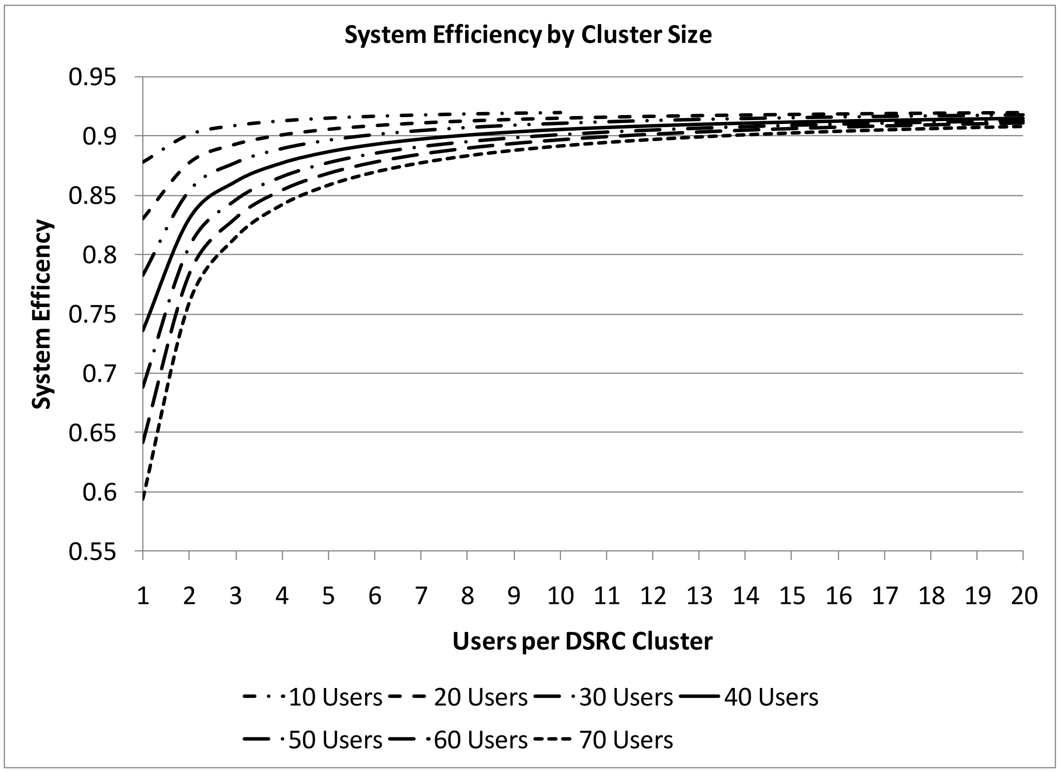

Figure 2.6: System Efficiency by Cluster Size

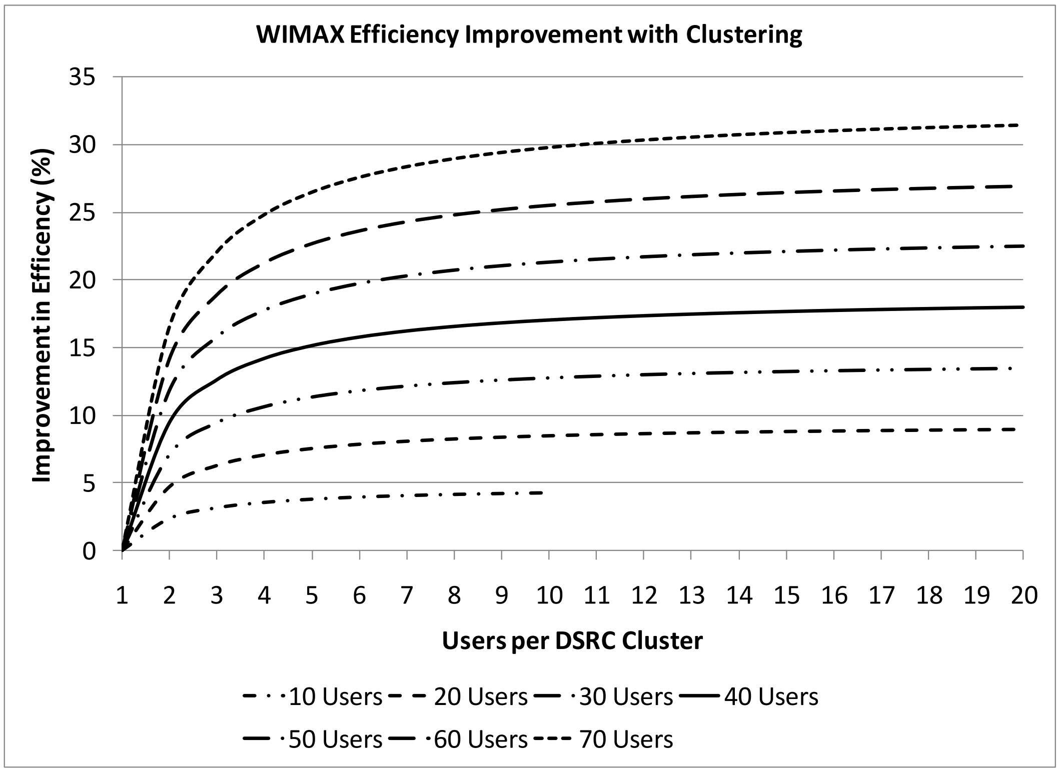

Figure 2.7: WiMAX Efficiency Improvement with Clustering

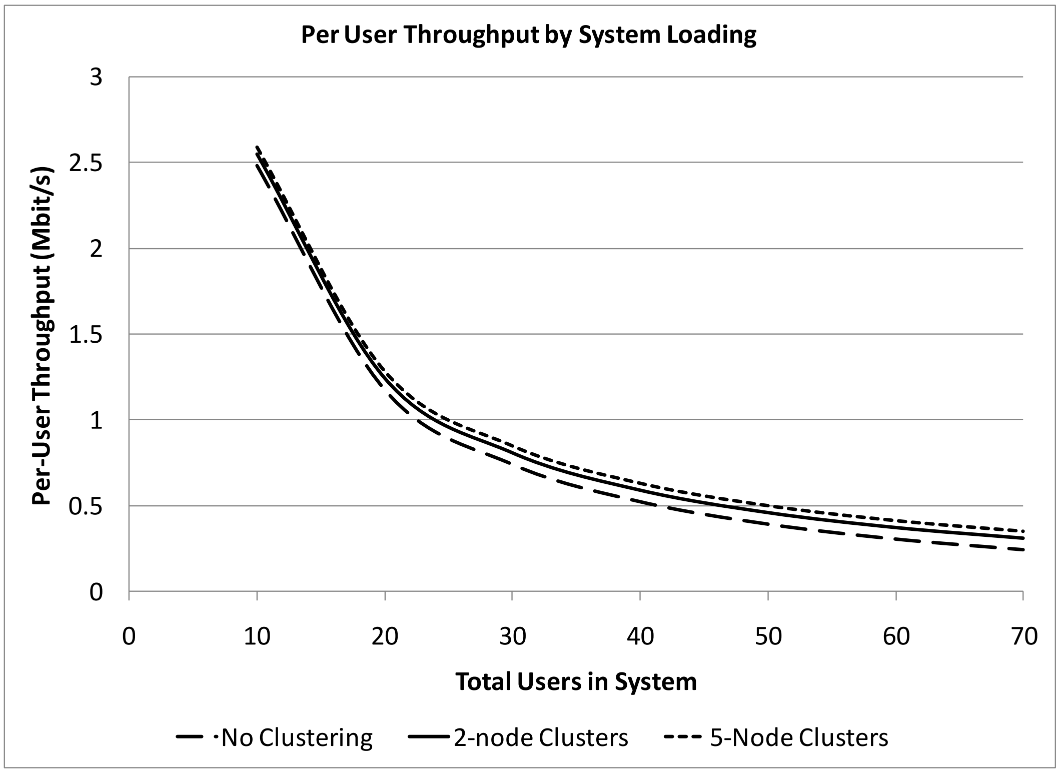

Figure 2.8: Per User Throughput by System Loading

2.7 Summary

This chapter began with an analysis of the overhead produced in each frame as a function of the number of active connections in the system. Verifying previous research, analysis using the S-OFMDA frame structure used in Mobile WiMAX found that the overhead in the system increased as a function of the number of connections, with the system capacity dropping to 60% with 70 active connections.

To address this issue, the concept of clustering and connection concentration was discussed. By grouping together nodes at a local level, bundling their traffic and transmitting through a designated relay node, the total number of connections across the WiMAX link can be reduced, while still providing connectivity for all nodes involved. The system was tested again at different loading levels with different levels of clustering. The results found that clustering was quite effective in reducing system overhead in the system, particularly when the system is heavily loaded. This resulted in a noticeable increase in the per-user bandwidth available in the system.

In the next two chapters, a system will be proposed to implement the clustering scheme described in this chapter. Chapter 3 will describe a combined WiMAX/DSRC system, in order to provide more efficient system access to vehicular based nodes. Chapter 4 will complete the system by proposing a network-layer protocol to provide mobile Internet access within the system.

Chapter 3: Proposed WiMAX/DSRC System

3.1 Introduction

The previous chapter explored the effect of frame overhead in a WiMAX system as the number of active connections within the system increased. The analysis confirmed previous research and found a rapid increase in the percentage of the frame consumed in system overhead, with upwards of 40% of the bandwidth consumed by overhead in a heavily loaded system with 70 active connections.

Connection concentration was found to be a possible solution to this issue. The proposed system structure was to use the short-range DSRC networking technology to group nodes into units known as clusters. A controlling node, known as the cluster head, manages all traffic in the cluster. It is also responsible for concentrating WiMAX traffic. The benefit to the WiMAX system with this proposed network topology was analyzed in the previous chapter. The improvement when nodes were clustered became immediately obvious.

In light of these results, this chapter and the next will describe a system to implement the proposed network topology. This chapter will describe the structure of the combined WiMAX/DSRC system. This includes a description of the system components, the protocols used for communication and a description of the system transactions.

The next chapter will complete the system by describing a network layer protocol that will allow for a network layer connection to the Internet to be established across the WiMAX/DSRC network. This will allow rapid mobility in the system as described in the principles of micromobility.

3.2 System Structure

The previous section discussed the capacity problems with WiMAX as the system became congested with many simultaneous connections. At the same time, it was discussed that Internet access over mobile ad-hoc networks formed by DSRC was found to be impractical for high bandwidth and low latency applications.

The solution developed in the previous chapter was a hybrid system, where vehicles in high congestion situations (major roads and highways) would form into ad-hoc clusters using DSRC.

The cluster heads could then be used to concentrate the signals and communicate the traffic over WiMAX and across a single tunnel connection to the base station node. To implement this system, vehicle nodes would be equipped with both a DSRC and a WiMAX radio.

This system proposes that members of the DSRC cluster accessing the network are fully registered nodes on the WiMAX network as opposed to the cluster heads functioning as an Internet access gateway. This was done for a few reasons. First, requiring node registration gives the system provider better control over the nodes accessing the system, through better access control, authentication and usage accounting. This makes, among other things, billing users accessing the WiMAX system easier. Second, by making use of the mobility features of WiMAX, nodes registered with the system can be tracked through mobility much easier. This makes uninterrupted handovers of connections a much smoother process.

Figure 2.5 in the previous chapter showed the overall layout of the system. On congested routes, nodes located close to one another would form clusters using DSRC. As seen in the diagram, they would pick one of the nodes within the cluster to become the cluster head. This node is responsible for communications

The underlying protocol used by the DSRC network to form the clusters is outside of the scope of this thesis, as ad-hoc clustering algorithms are a research area into themselves. Previous research [**20**] has suggested some potential methods of electing cluster heads within DSRC networks, as well as demonstrating the viability of this approach to vehicular networks.

Figure 3.1 shows the overall structure of the proposed system. Four node types are identified, as well as their interactions with one another. The three nodes providing the combined WiMAX/DSRC system (SS, RS and BS) will be described in this chapter, with the IAG node and the user applications being described in the next chapter.

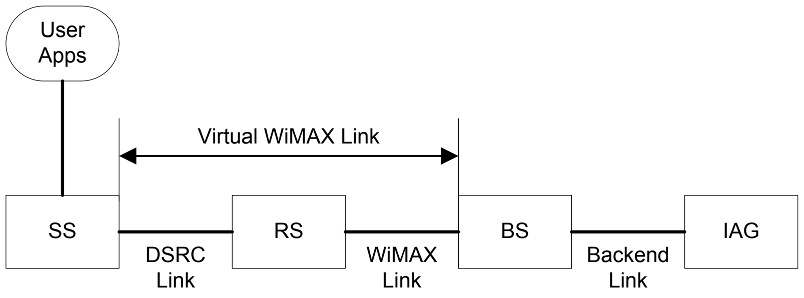

Figure 3.1: System Structure

3.3 System Nodes

There are four basic nodes defined in this system: the Subscriber Station (SS) node, the Relay Station (RS) node, the Base Station (BS) node and the Internet Access Gateway (IAG) node. These each represent a different class of node within the system, with different responsibilities and capabilities.

Subscriber Station Node

The Subscriber Station (SS) node is the end user in the system, and is a concept carried over from the WiMAX system. Within this system, the SS node is a user in a mobile vehicle accessing Internet services through the network. The SS node is a member of a DSRC network and communicates through the cluster-head node.

Relay Station Node

The relay station (RS) node is responsible for relaying information between the DSRC network and the WiMAX network. It is based heavily on the non-transparent relay node described in the draft IEEE 802.16j standard. It functions as a pseudo base station to nodes operating under it. It then transfers the data from these users to the cell's base station across a tunneled traffic link.

Base Station Node

The base station (BS) node performs the functions of the BS node outlined in the 16j specification. Within a WiMAX cell, the BS node is responsible for allocation of resources and handling authentication and handovers.

Internet Access Gateway Node

The Internet access gateway (IAG) node functions as the gateway between the WiMAX provider's network and the greater Internet. The functionality of this node will be described in the following chapter.

3.4 Subscriber Station (SS) Node

The subscriber station node represents a subscriber using the WiMAX system. In the context of the proposed system, the SS node is a vehicle based node that is part of a DSRC cluster. It communicates to the WiMAX network through a link with the DSRC cluster head.

The SS node structure is described in Figure 3.2. The SS node contains two radios – a DSRC radio and a WiMAX radio. The WiMAX radio is not used in the proposed system, as the SS node's communication with the WiMAX network is done through the relay station node. The details of the DSRC link are described in Section 3.8.

While the SS node in the described system only connects to the base station through the relay station node, it is possible for the node to connect directly. As mentioned in Chapter 2, purely ad-hoc systems suffer when there isn't enough node density to ensure constant routing. In situations where node density doesn't allow for clustering, SS nodes can use the WiMAX radio to connect with the base station node directly. This is the current Mobile WiMAX network structure and will not be described in further detail. Through use of the protocols described in this chapter and the next, it will be possible for nodes to switch between being served in a DSRC cluster and being served through a direct connection and still maintain their network connections.

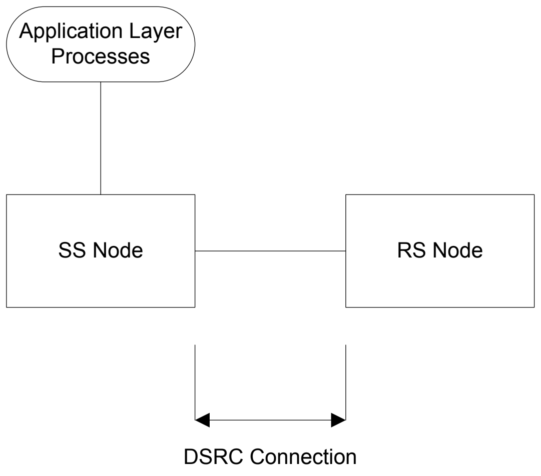

Figure 3.2: SS Node Structure

The SS node uses a single network link. The SS nodes communicate with the cluster head in the DSRC network (the RS node) across a DSRC link. The details of this link are described in Section 3.8. The network entry process between the SS node and the RS and BS nodes is described in Section 3.9.2.

3.5 Relay Station (RS) Node

The relay station node is based on the relay station described in the IEEE 802.16j draft specification. However, rather than relying on in-band or out-of-band WiMAX for relaying the messages, the RS node uses DSRC as the relaying medium.

The RS node and the SS node described in the previous section are identical in construction, with the RS node simply being an SS node that was elected to being a relay by the underlying DSRC protocol. The method by which the node is selected is outside of the scope of this research.

The RS node has the structure shown in Figure 3.3. On the DSRC interface, it accepts connections from multiple SS users. On the WiMAX interface, it establishes an uplink connection with the base station node.

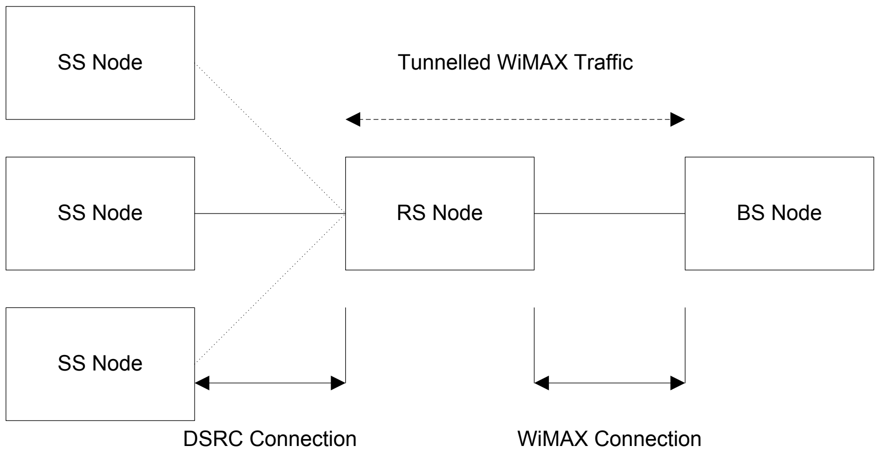

Figure 3.3: RS Node Structure

The RS node has two sets of communications. First, it is connected directly across the WiMAX network to the servicing base station (BS) node in the WiMAX cell. This link is used to communicate both configuration network for the RS node, as well as for tunneling traffic to and from the SS nodes served by the RS node. This link is described in Section 3.7. The connection establishment process between the RS node and its serving BS node is described in Section 3.9.1. As well, each individual SS node within its cluster establishes a connection with the RS node over the DSRC link. This link is described in Section 3.8.

3.6 Base Station (BS) Node

The base station node is responsible for the allocation of resources in the WiMAX network, authentication and accounting of users, and coordinating network handovers between base stations. The BS node is also responsible for forwarding application layer data to the responsible application providers. Within the context of the proposed system, this includes forwarding Internet access data to the agent responsible for coordinating the network access.

The SS node structure is shown in Figure 3.4. As described in the 16j amendment, the BS node manages connections from relay stations, as well as direct connections from subscriber stations. As the primary controller for the WiMAX cell, it also accepts relayed connections from SS nodes (as described in the 16j amendment).

The BS node is also connected to the backend network set up by the WiMAX service provider. This network is used for many purposes, such as coordination between BS nodes for handovers and to authenticate users against a master authentication server. Another purpose for this connection is to provide services to the user, such as Internet access. One of these services will be described in the following chapter and will involve the BS nodes connecting with a forth node, the Internet access gateway.

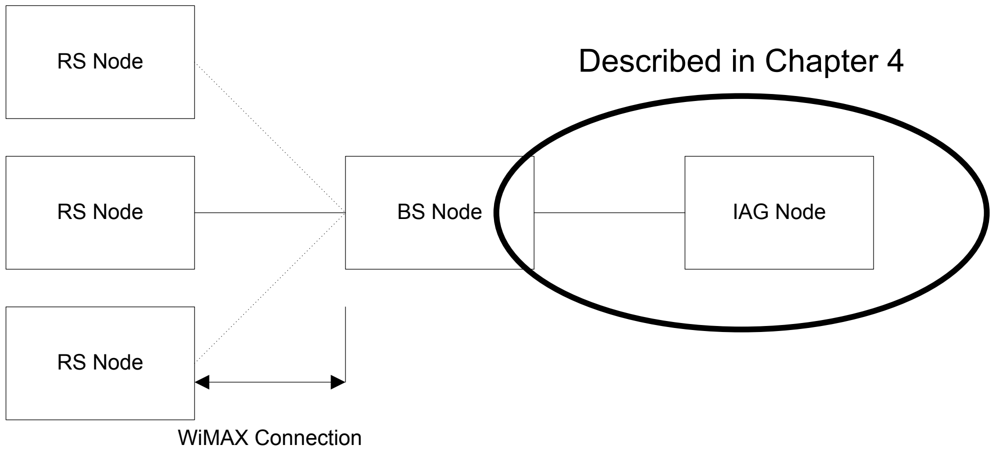

Figure 3.4: BS Node Structure

The communication between the BS node and connected RS nodes is described in Section 3.7. The backend communication to the IAG node will be described in the following chapter.

3.7 WiMAX Connection

As defined in the IEEE 802.16e specification [**6**], there are two main classes of communications: management messages and user data. The management messages are used to establish communications and update the link as circumstances change. The user data messages are used to transmit application layer data on behalf of the clients using the system. Within the context of the proposed system, this represents L3NP packets and Internet traffic.

The IEEE 802.16j specification [**7**] adds another class of messages for traffic between the relay station and the base station called relay messages. Relay messages allow a relay station to group together traffic to and from the base station by subscriber station nodes served by the relay station, and transmit them using a single connection (CID), making more efficient use of the WiMAX bandwidth.

3.7.1 Generic MAC Header

Both management and user data PDUs use a common MAC header. As defined in the IEEE 802.16 specification, this header is shown in Figure 3.5. Table 3.1 lists details on the different fields in this header.

LSB

HT (1)	EC (1)	Type (6)	ESF (1)	CI (1)	EKS (2)	Rsv (1)	LEN (MSB) (3)
LEN (LSB) (8)			CID (MSB) (8)				
CID (LSB) (8)			HCS (8)				

Figure 3.5: Generic WiMAX MAC Header

Table 3.1: Generic MAC Header Fields

Name	**Length (bits)**	**Description**
CI	1	CRC Indicator 1 = CRC is appended to the end of the PDU 0 = No CRC is used
CID	16	Connection Identifier
EC	1	Encryption Control 1 = Payload is encrypted 0 = Payload is not encrypted
EKS	2	Encryption Key Sequence
ESF	1	Extended Subheader 1 = Using extended subheader 0 = Extended subheader absent
HCS	8	Header Check Sequence
HT	1	Header Type (use 0 for generic header) 1 = Bandwidth Request Header 0 = Generic MAC Header
LEN	11	Length Length of the MAC PDU in bytes (including header and CRC, if used)
Rsv	1	Reserved for future use. Set to 0.
Type	6	Flags to indicate the type of payload attached

The different message types are identified by the connection IDs (CIDs) used in the message. During the initial connection process, nodes will be assigned several CIDs. Some (the basic and primary management CID) are used for management messages. RS nodes get assigned CIDs for tunneling, while SS nodes will request CIDs for service flows. These different message types will be described in this section.

3.7.2 Management Messages

As the name suggests, the management messages are used to manage the user sessions on the WiMAX network. This includes tasks such as network entry, ranging (initial and periodic), bandwidth requests and handover requests. Specific CIDs are assigned to each user within a WiMAX cell for management purposes. These include the assigned basic CID (BCID) and the assigned primary management CID (PMCID), as well as a default CID (usually 0) used during the original network entry. The structure of management messages is shown in Figure 3.6. Some of the management messages used heavily during a typical WiMAX session (and seen in the proposed system) include the ones listed in Table 3.2.

LSB

Generic MAC Header	**Management Message Type**	**Management Message Payload**	**CRC (If Used)**
(See Figure 3.5)	(1 byte)	(Variable Size)	(4 bytes)

Figure 3.6: WiMAX Management Messagee

Table 3.2: WiMAX Management Message Types

Message Name	Type	Spec Definition	Description
RNG-REQ	4	6.3.2.3.5	A message from an SS/RS node requesting ranging information.
RNG-RSP	5	6.3.2.3.6	The reply from the BS/RS with ranging information and basic/primary management CID parameters.
REG-REQ	6	6.3.2.3.7	A request from an SS/RS node to finalize registration on the WiMAX network.
REG-RSP	7	6.3.2.3.8	A reply from the BS node confirming or denying registration. Tunnel CIDs for RS nodes are provided at this point as well.
PKM-REQ/ PKM-RSP	9/ 10	6.3.2.3.9	Key sharing for authentication and securing of the connection.
DSA-REQ	11	6.3.2.3.10	A request from an SS node to open a service flow with the BS node.
DSA-RSP	12	6.3.2.3.11	The BS reply. If successful, includes the CID for the service flow.
DSA-ACK	13	6.3.2.3.12	Acknowledgement by SS of the new service flow.
MOB_BSHO-REQ	56	6.3.2.3.52	A request from the BS node to an SS node to initiate a handover process.
MOB_MSHO-REQ	57	6.3.2.3.53	A request from the SS node to initiate a handover process.
MOB_BSHO-RSP	58	6.3.2.3.54	A reply from the BS node confirming initiation of the handover process.
MOB_HO-IND	59	6.3.2.3.55	Indication from the SS node that the handover process is complete.

More details on management messages can be found in the IEEE 802.16 specifications.

3.7.3 User Data Messages

User data is transmitted across the WiMAX system in structures known as service data units SDUs). This data comes from applications operating at a higher protocol level, such as network layer traffic. The Layer 3 Tunneling Protocol (L3TP) frames described in the next chapter is an example of this kind of traffic. Frames containing SDUs are structured like the one in Figure 3.7.

User data is transmitted across service flows. IEEE 802.16 defines these as “a MAC transport service that provides unidirectional transport of packets either to uplink packets transmitted by the SS or to downlink packets transmitted by the BS”. During the connection establishment process with the BS node, the SS node will be assigned a Service Flow CID (SF-CID) associated with this service flow. This flow will be linked to a network service and both the SS and the BS nodes will forward it properly.

In the case of the proposed system, the SS and BS nodes will transmit data from the upper layer network protocol using this service flow. The BS node will forward these SDUs on to the IAG for processing. The SS node will forward service flow traffic to the upper layer process, which will be responsible for providing Internet access for user applications. This process will be described in more detail in Chapter 4.

LSB

Generic MAC Header (See Figure 3.5)	**Service Flow Data** (Variable Size)	**CRC (If Used)** (4 bytes)

Figure 3.7: WiMAX User Data Message

3.7.4 Relay Messages

Relay frames are special frames defined in the 16j draft. They allow a relay station to group Protocol Data Units (PDUs), which are WiMAX frames in this context, from multiple users into a single connection between the relay station and the base station. This reduces the number of active connections in the system, which in turn lowers the system overhead produced by controlling the individual connections. Likewise, the base station will group PDUs destined for users served by the relay station into this single connection.

According to the 16j specification, the relay frames take the format of Figure 3.8. The relay header itself is shown in Figure 3.9. The fields in the WiMAX Relay Header are described in more detail in Table 3.3.

Relay Header	User PDU #1	User PDU #2	…

Figure 3.8: WiMAX Relay Frame Structure

HT (1)	Rsv (1)	RMI (1)	Rsv (7)	Priority (3)	LEN (MSB) (3)
LEN (LSB) (8)			CID (MSB) (8)		
CID (LSB) (8)			HCS (8)		

Figure 3.9: WiMAX Relay Header

Table 3.3: WiMAX Relay Header Fields

Name	Length (bits)	Description
RMI	1	Relay Mode Indication 1 = Use Relay MAC header 0 = Use GMH
CID	16	Connection Identifier Tunnel CID or basic CID of the RS
Priority	3	Priority of the tunneled MPDU
HCS	8	Header Check Sequence
HT	1	Header Type (use 0 for relay header) 1 = Bandwidth Request Header 0 = Relay Header
LEN	11	Length Length of the MAC PDU in bytes
Rsv		Reserved for future use. Set to 0.
Type	6	Flags to indicate the type of payload attached

3.8 Virtual WiMAX Link

The virtual WiMAX link is a key component to the proposed system. The system allows out-of-band relaying of a WiMAX signal across another technology, in this case DSRC, which is more appropriate for forming clusters of vehicles. The virtual WiMAX link will attempt to implement as much of the WiMAX protocol as possible. This allows SS nodes to function as full members of the WiMAX network.

3.8.1 DSRC Link

As discussed in Chapter 1, DSRC is a protocol for vehicle-based communications. At the MAC level, DSRC is based on the IEEE 802.11p standard. Based on the popular IEEE 802.11a WiFi standard, this is a contention-based protocol for short range communications [**3**].

This thesis does not specifically lay out the protocol and clustering technique used for the link layer connection across the DSRC network. The proposed system assumes that the clusters are successfully formed and that the cluster members know which node is the cluster head and can communicate with them. The specifics of this link are contingent on the underlying DSRC network structure. For the sake of simplicity, the proposed system assumes that the nodes communicate using TCP/IP, which makes the DSRC network flexible for multiple applications. MAC layer communications would also be possible, depending on how the underlying clustering protocol works. Definition of a MAC layer protocol will be left to research surrounding the underlying clustering protocol for DSRC.

In the proposed system using TCP/IP, all nodes in the system have an IP address. The allocation of the addresses is done by the underlying DSRC protocol. SS nodes are aware of the IP address of the serving RS node. The RS node offers a WiMAX service at a known port. When an SS node wants to communicate with the WiMAX network, it will connect to the service on the RS node. Once this TCP/IP connection is established, WiMAX messages in the form described in Section 3.8.2 are sent across the connection.

3.8.2 Packet Structure

The virtual WiMAX frames follow the WiMAX frame structure as closely as possible. Communications between the BS and RS nodes will consist of WiMAX packets encapsulated within the underlying protocol, as shown in Figure 3.10.

LSB

DSRC MAC Header	**TCP/IP Headers (If Used)**	**Encapsulated WiMAX Packet**	
		WiMAX MAC Header	**Message Body**

Figure 3.10: Virtual WiMAX Frame Structure

3.8.3 Connection Differences with WiMAX

As the WiMAX session is being established over a non-WiMAX medium, there are some minor differences from a typical WiMAX connection that have to be addressed. These stem from the fact that the virtual WiMAX system is not accessing the WiMAX network at a link layer, but is instead being virtually carried on top of another network technology.

During a typical WiMAX network entry, the node listens to the channel to pick up system information, such as the channel allocations, the Base Station ID (BSID). This is not possible with the virtual WiMAX system, as it is not using a broadcast medium. To replace this functionality, the new SYNC command was created, using a management message type of 255. This command provides the initial connection information. The message has the structure seen in Figure 3.11, with the fields described in more detail in Table 3.4.

LSB

Generic MAC Header (See Figure 3.5)	**0xFF** (Message Type for SYNC)	**System Version** (2-4 bytes – TBD)	**BSID** (4 bytes)

Figure 3.11: Virtual WiMAX Frame Structure

Table 3.4: SYNC Management Message Fields

Field	Description
Version	The protocol version of the "Virtual WiMAX" system. This ensures that the client software is compatible with the protocol used by the RS node.
BSID	The base station ID of the RS node. Used for the network entry messages, etc.

The virtual WiMAX system uses an abbreviated ranging process during the network entry, as there is not the iterative transmission of power settings and signal strength information. Instead, the ranging command is only used once and the reply contains all the important information supplied in a WiMAX ranging session, including the basic and primary management CIDs.

Finally, since the WiMAX physical layer is not used, no bandwidth reservation is used. Instead, WiMAX frames are transmitted serially across the underlying DSRC network connections.

3.9 Network Entry Process

The following sections describe the network entry process of the proposed system. This describes both the entry of the RS node into the system, as well as an SS node entering the system through the RS node.

3.9.1 Establishing RS-BS Connection

The connection handshake process follows the process laid out in IEEE 802.16j. This process authenticates RS nodes with the WiMAX network and sets them up to accept connections with SS nodes, which will be tunnelled and processed by the network. The entry process is shown in Figure 3.12, and will be described in detail in this section.

The BS node maintains a routing table of active connections within the WIMAX cell. This includes SS and RS nodes connected directly through the WIMAX interface, as well as SS nodes connected indirectly through RS nodes registered with the BS. The information for RS nodes includes the assigned CIDs (B-CID/PM-CID/MT-CID/T-CID) and the assigned RS BSID. The information stored for SS nodes includes the assigned CIDs (B-CID/PM-CID/SM-CID) and the currently registered RS node. As well, each entry stores the current status of the connection and encryption information (public key, current session key, etc).

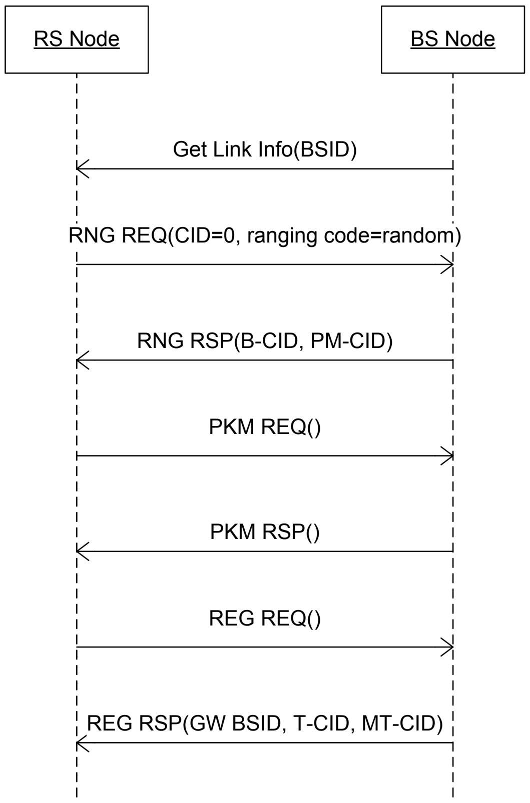

Figure 3.12: RS Network Entry

Get Link Information

This is the traditional method of entry for WIMAX nodes over an S-OFDMA air interface. This process is used to passively retrieve channel parameters about the cell.

The RS node scans to find the closest BS node. It then synchronizes with the WIMAX frames and listens for the channel MAPs and channel descriptors. The BSID of the BS is determined. It also determines the appropriate allocation within the frame to send the initial registration message.

Ranging (RNG)

The ranging process is the first communications between the RS node and the BS node it is registering with. This process is conducted according to the normal access process specified in IEEE 802.16e and IEEE 802.16j.

One of the parameters transmitted with the RNG REQ message is the RS's MAC address. It is advisable that the WIMAX radio have two MAC addresses – one for SS functionality and one for RS functionality. Since each RS node is also an SS node, this allows the RS node to separate the subscriber node functionalities (such as the end user applications in that vehicle) from the WiMAX/DSRC gateway functionality. This makes promotion to/demotion from a RS node easier and to allow the BS node to track user traffic separately from RS traffic. This will result in user data travelling across the same tunnel as the rest of the cluster. In this configuration, the SS functionality in the RS node would occur as a virtually connected SS connection. As mentioned below in the registration section, a RS-specific MAC allows the BS to determine if the node connecting is a RS or SS node.

Privacy Key Management (PKM)

The PKM process performs the initial authentication of the BS node using public keys. It also establishes the session keys for encrypted communications between the RS and BS nodes. This will be used by the RS node to secure communications of tunneled information. As WiMAX is used over an insecure broadcast medium, it is good practice to encrypt the transmissions.

Registration (REG)

This is the final step in the network entry for the RS node. At some stage, it was determined that the node is a RS. This may require modification of a field (adding a new TLV-encoded parameter or modifying one of the existing ones to have an additional value for RS nodes), or it may be determined by the RS MAC used during the initial ranging. This will be defined in the final release of the IEEE 802.16j amendment.

The BS node provisions three values required for the RS role at this stage. The first is a valid BSID to be used by the gateway for accepting clients. This is the effective "BS" that each client will be using, even though all traffic is ultimately forwarded by the RS to the BS node responsible for controlling a cell. This BSID is unique within the WiMAX system.

The other two values are the management tunnel CID (MT-CID) and tunnel CID (T-CID). These values are specified by IEEE 802.16j to provide the functionality for tunnelling of SS traffic data between the RS and BS nodes. These CIDs will indicate to the BS that encapsulated traffic is being transmitted (as opposed to management messages between the BS and RS nodes).

At the completion of the handshaking process, the RS node has been assigned a number of parameters. From the initial ranging, it has a basic CID and primary management CID number. After the network registration, it is also assigned a BSID value, as well as the tunnel and management tunnel CIDs.

3.9.2 Establishing SS-RS-BS Connection

This section details how an SS node enters the WiMAX network, across a DSRC connection and using an RS node as its network entry point. This process is shown in Figure 3.13 and will be described in detail in this section. This is based on the client network entry process described in IEEE 802.16j.

The RS node maintains a forwarding table to track the connections across the WiMAX/DSRC gateway interface. This enables the RS node to properly route traffic as it crosses through the gateway interface, as well as tracking the status of the connections. The BS node maintains a table of all SS connections in the system, including the RS node currently serving each one.

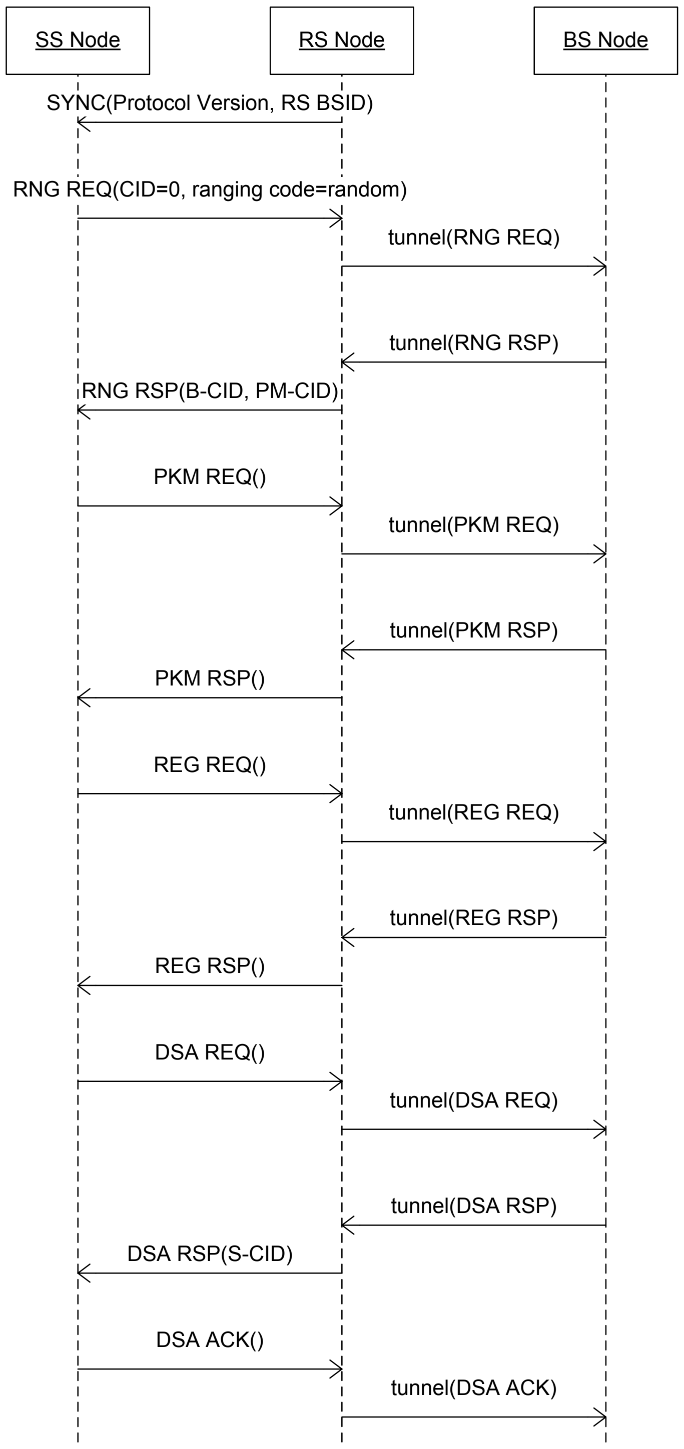

Figure 3.13: SS Network Entry

Connection Information (SYNC)

As mentioned in Section 3.8, the virtual WiMAX connection uses a substitute message (SYNC) to obtain initial connection information from the RS node. This message is sent upon initial connection between the SS node and the RS service on the RS node.

A new entry in the RS routing table is also created. Upon initial connection, this connection is empty (other than perhaps physically connection information, version number, physical address, etc.)

Ranging (RNG)

This process is conducted similar to the ranging process described in IEEE 802.16j. As the information is not being transmitted across a physical WIMAX link, message fields such as power control can be ignored. This also shortens the ranging process, which normally requires several iterations across a WiMAX physical layer.

In the ranging request message, the SS prepares a WIMAX control packet with a CID of 0. Within the RNG request, a random ranging code is selected. Other information transmitted includes the WIMAX MAC address of the SS node. This is the MAC address of the WIMAX radio, which isn't being used for the DSRC communications but will be used if the node later directly connects with the base station node.

When this packet is received at the RS, it is recognized as a ranging packet by the CID of 0 (as well as the command message packet header). The RS updates the routing table entry with the ranging code, and then forwards it along in a tunneled connection.

The packet is received at the BS and is processed like any SS entering the system, as defined in the IEEE 802.16j specification. As the packet is forwarded, message fields such as power settings are once again ignored. The BS provisions a B-CID and PM-CID and creates a new entry for the SS in its internal routing table, as is the assigned RS for that node. The reply message (RNG RSP) is sent back across the management tunnel to the RS.

When the reply is received at the RS, the entry is looked up in the routing table by the ranging code value. The ranging code value is removed and replaced with the B-CID and PM-CID values in the ranging reply. The message is then forwarded along to the SS node.

The message is finally received back at the SS node. The B-CID and PM-CIDs are used for further communications.

Privacy Key Management (PKM)

This stage ensures that the SS node is authorized to use the system. It involves a key exchange between the SS and BS node. If the authentication is accepted, the BS node replies back with an authentication reply message. This includes the session key, sequence number, and timer information, encoded in the SS's public key. The details of this process are described in the IEEE 802.16e standard.

Registration (REG)

After being authenticated, the node registers on the WiMAX network using the REG commands. This finishes the initial handshake process. At this point, the node is ready to send and receive traffic across the WiMAX network.

Provisioning of Service Flow

Once the system is registered, it needs to obtain a service flow for transporting the higher level network data. There are two methods that this can be accomplished with WiMAX. WiMAX systems can support "managed connections", where IP support is intrinsic to the connection. This IP configuration isn't optimized for mobility, where users will get a new IP upon handing over and existing IP connections are broken. The other option is to provide unmanaged connections. For an Internet access perspective, this unmanaged connection can be made to resemble a simple serial-like link layer, which is ideal for the PPP-like connection described in the following chapter.

The service flow is provisioned using the DSA (Dynamic Service Add) series of commands. These specify the QoS/service parameters describing the Internet access link service requested. After the DSA handshake is complete, the node will be assigned a service CID (S-CID), which is then ready for use for Internet access.

Upon receipt of the DSA RSP message (with the assigned CID), the RS node will update its routing table with the S-CID value. It also updates the status of the connection to indicate that the SS node is registered and ready to handle user traffic. This assigned CID is known throughout the rest of this paper as the Service Flow CID (SF-CID), although the WiMAX standard also refers to it as a Transport CID.

At this point, communications between the individual SS nodes and the BS node through the RS node has been established. This service flow will be used for the network layer protocol described in the following chapter.

3.10 Handover Process

Handovers occur when a node leaves one service area and enters into another. When going through a handover, the node has to connect and authenticate with the new service area and then transfer its session from the old one.

In the context of the described system, there are several kinds of handovers that will occur. These are divided up into SS node handovers and RS node handovers. RS node handovers are somewhat more complicated, as it involves the indirect handover of several different connections at the same time.

The SS is subject to three handover conditions. The first is moving from one DSRC cluster to another (RS - RS). The other two involve entering a cluster mode (BS- RS) or exiting cluster mode (RS -BS). Finally, the RS and BS nodes will experience a handover condition when a mobile RS node hands over from one BS node to another (BS-BS).

In all cases, we are assuming that cross-layer information is not available (eg the system is not aware of an impending handover situation), which will result in an abrupt termination of connections. These handover scenarios are based on the ones described in the 16j amendment, updated to reflect the virtual WiMAX interface.

3.10.1 SS performing a RS - RS handover

This handover occurs when a node leaves one DSRC cluster and enters another. This can be a vehicle going faster or slower than the cluster head, or a new node is elected as the cluster head. Section 6.3.22.2 of the IEEE 802.16e standard discusses the way standard Mobile WiMAX performs a handover. This begins with an SS node issuing a MOB_BSHO-REQ command to alert the current BS node that it is leaving. In the DSRC link, this will not be possible due to the assumption of no cross-layer information and, instead the link termination will be abrupt. On the other hand, the termination of the underlying link between the SS and RS node will provide confirmation that the connection is terminated.

When a handover occurs, the underlying connection between the SS and the RS will be broken. Both nodes will be aware of the connection being terminated. Both nodes will go into a "hand-over" state and will start a timer. If the hand-over doesn't occur in a timely manner, the connection will be terminated. All connections will be terminated and all system resources (such as CIDs) will be returned to the pool for reallocation.

When the SS node connects to the new RS, it will go through the network entry process in Section 3.8.2. The difference is that the node supplies a "serving BSID" value in its RNG REQ message. This alerts the new RS node that the SS node is handing over from another relay station. This information is forwarded along to the serving BS node. If the serving BS node is the same (one RS to another within the same BS cell), the information is updated in the BS's routing tables. If the node is changing serving BS nodes, the new BS node will obtain session information from the old BS node, as the RS node's BSID is unique and routable to a specific BS node.

3.10.2 SS Performing BS-RS Handover

There are two special cases in the client handover – a client that is directly connected to the BS joining the DSRC cluster system, and a client leaving the clustered system for a direct WiMAX connection. In realistic terms, this could represent a client entering or leaving the freeway or a major road for a less populated side road.

When a node is entering a DSRC system, it performs the handover as described in Section 6.3.22.4 of the IEEE 802.16j proposal. However, instead of performing a full scan, it begins the connection with the RS node while maintaining a connection with the BS node using the WiMAX radio. Once the connection with the RS node is complete, it finishes the handover and terminates the direct WiMAX connection with the BS node.

3.10.3 SS Performing RS-BS Handover

This handover occurs when the SS node leaves a cluster situation and starts direct communication with the BS. Like with an RS-RS handover, this will likely be abrupt, giving the node no time to prepare for the handover.

Much like an RS-RS handover, the node starts by establishing a connection with the BS node across the WiMAX interface using the network entry process described in Section 6.3.9 of the IEEE 802.16e specification. Like with the RS-RS handover, the SS node supplies a "serving BSID" value in its RNG REQ. This allows the BS node to carry over the session and continue any established sessions uninterrupted.

3.10.4 RS performing a BS-BS Handover

As the RS node itself is going to be mobile, it will also be subject to frequent handovers. This is probably the most frequent handover that will be seen in the proposed system. The mobile relay station handover process is described in Section 6.3.22.4 of the IEEE 802.16j proposal. The CIDs for the SS nodes will be updated according to the “MS CID” process, which was still being developed at the time this thesis was being written.

3.11 Summary

Following the analysis in the previous chapter about the benefits of connection concentration in WiMAX, this chapter proposed a system to combine WiMAX with clustered DSRC to provide more efficient use of WiMAX bandwidth. The proposed system took advantage of DSRC’s focus on vehicular applications and the nature of heavy traffic in order to form ad-hoc clusters to concentrate WiMAX connections. The chapter described the main nodes in this interaction and gave a description of the protocols used in the connections between them. Finally, the processes for both network entry and handovers were discussed in detail.

In the next chapter, the functionality provided by the combined WiMAX/DSRC system will be further extended. A network layer protocol will be added on top of the system to provide mobile Internet access for user applications. This protocol is designed to be robust to the high level of mobility present in the proposed WiMAX/DSRC system, being able to maintain Internet connections across frequent handovers. At the same time, it ventures to use the principles of micromobility to keep overhead at a minimum.

Chapter 4: Proposed Network Layer Protocol

4.1 Introduction

The previous chapter described a combined WiMAX/DSRC approach to providing a link layer connection to a WiMAX network across two network technologies. DSRC was used to cluster nodes together, with the cluster heads communicating across WiMAX in a manner similar to the relay station described in the 16j amendment. Chapter 3 laid out in detail how this protocol was going to be implemented in order to provide subscriber station nodes access to the WiMAX network.

The network layer protocols described in this chapter are the second component in a complete network access solution. This application layer functionality operates on top of the underlying WiMAX/DSRC architecture. By adding this network layer on top of the underlying link layer, subscriber stations can establish and maintain a stable connection to the Internet. This provides the stable connection required for applications like streaming audio, video and Voice over IP (VoIP).

This section includes a description of the communications between the Base Station and Internet Access Gateway nodes, as well as a description of the network layer protocol used to provide this service.

4.2 System Components

Building on the system structure described in the previous chapter, Figure 4.1 shows the entire system including the additional Internet Access Gateway (IAG) node, as well as the application layer software. This section explains the purpose of the IAG node and the protocol used for communication with the Base Station (BS) node. The Access Concentrator (AC) and Network Server (NS) application layer modules are described, as well as the Layer 3 Network Protocol used to communicate between them.

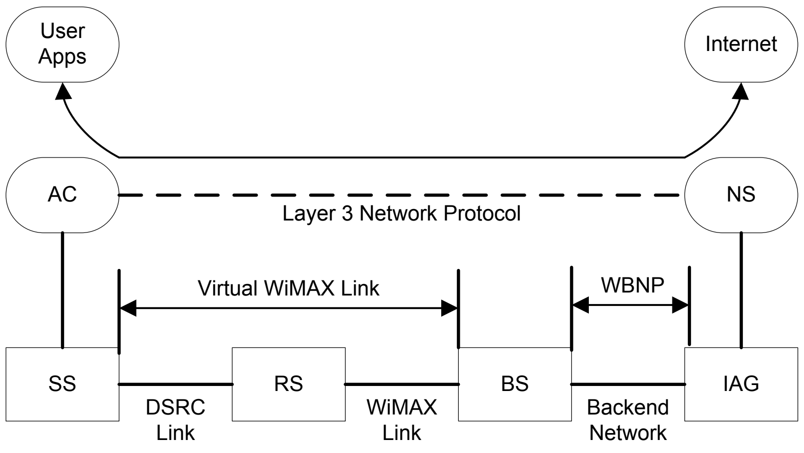

Figure 4.1: Overall System Structure

Internet Access Gateway

The Internet Access Gateway (IAG) node functions as a gateway between the local network and the global Internet. Following the principles of micromobility described in Chapter 1, the IAG allows clients within the inner network and operating across the WiMAX network to maintain their Internet connections while being highly mobile within the system, migrating from relay station to relay station and from base station to base station.

The IAG node is connected to the base station nodes in the system across the backend network. Using the network protocol described below, the connections are established during initial network entry by the BS. The system described in this thesis proposes a single IAG for the entire network in order to simplify routing. This is for simplicity sake, although the mobility scheme described below does rely on a single gateway to the Internet. This node will be described in detail in Section 4.3.

Access Concentrator Module

As part of the Layer 3 Network Protocol described in Section 4.7, the Access Concentrator (AC) module provides the client services for SS nodes. It operates as application level software on the SS node, using the underlying WiMAX and backend network. The AC module uses the service flow established during network entry to communicate to the IAG node. All AC traffic will be transmitted by the SS node using the SF-CID. Likewise, all traffic received by the SS with the SF-CID will be sent to the AC module for processing. This node will be described in detail in Section 4.6.

Network Server Module

The Network Server (NS) module operates as an application layer service within the IAG node. It functions as the endpoint for the L3NP communications. The NS module is responsible for assigning Session ID (SID) numbers, performing session setup (using a modified version of the LAP) authenticating users (possibly utilizing a RADIUS style authentication server and protocols like PAP), and establishing network connections (using the IPCP). This node will be described in detail in Section 4.5.

Layer 3 Network Protocol (L3NP)

A custom protocol, called the Layer 3 Network Protocol (L3NP), is used for communication between the end user and the network gateway. L3NP is a modified version of the Point-to-Point Protocol (PPP), with the addition of support for mobility. Using the principles of micromobility, users can travel through the WiMAX system without having to re-establish their network connections. This protocol will be described in detail in Section 4.7.

WiMAX Backend Network Protocol (WBNP)

The WiMAX Backend Network Protocol (WBNP) is used for communications between the BS node and the IAG node. This simple protocol transfers L3NP packets between BS nodes and the IAG node. This protocol will be described in detail in Section 4.8.

4.3 Internet Access Gateway (IAG) Node Functionality

The IAG node is responsible for hosting the Network Server (NS) module and providing Internet access to clients in the WiMAX system. It functions as a micromobility gateway node, as described in Section 1.4, providing the access between the mobile local network and the Internet.

The IAG node is connected to each BS node in order to accept Internet traffic from users served by those nodes. This connection is established across the service provider’s backend network, the network used for BS-to-BS communication as well as communicating between the BS and provided services (such as Internet access and voice traffic).

The IAG node provides an Internet access service, which the WiMAX BS nodes offer as a service flow. SS nodes access this server by opening up a service flow during the network entry process detailed in Section 3.9.2, and then using the L3NP protocol described in Section 4.7 to open a connection. The L3NP protocol also describes how SS nodes should behave when they migrate through the WiMAX network.

4.3.1 Routing

The IAG node maintains three routing tables. Some of these tables are also used by the Network Server (NS) module described in Section 4.5. The primary principles used for routing include the Session ID assigned by L3NP to a given SS node, as well as the BSID for the currently serving BS and the L3NP Service Flow CID (SF-CID) currently assigned to that user. These three pieces of information are enough to uniquely identify the user and the current route to the user.

Using the L3NP protocol described in Section 4.7, individual SS nodes establish a connection with the IAG node across the combined DSRC/WiMAX network. The L3NP connection in the IAG node is handled through the Network Server (NS) module described in Section 4.5. However, the IAG node maintains the routing tables.

Each user within the L3NP system will be assigned a unique identifier called a Session ID (SID). The assigned SID is stored within the NS, along with the current status of that use (connection state, session keys, etc). This SID will be used by the user for the duration of their session with the system.

The backend network communication between the BS nodes and the IAG node is used to transfer L3NP frames between the WiMAX network and the IAG node. The communication between these nodes is accomplished with the WiMAX Backend Network Protocol (WBNP) described in Section 4.8. This communication ties each base station with a unique Base Station ID (BSID). The WBNP protocol transmits the CID assigned to the user for L3NP with each packet. The CID delivered with the packet, as well as the BSID implicit with the connection, forms the reverse route back to the user.

When traffic comes in from a user, the SID is used to look up the user entry. The BSID and CID of the received packet are compared to the values within the table. If they are different, it means the node has handed over to a new node and the values are updated. When the IAG is routing Internet traffic from the NS node, it will use the routing tables to look up the BSID and CID of the user. The traffic is transmitted to the proper BS node using the WBNP protocol.

4.4 Base Station (BS) Node Functionality

In the previous chapter, the BS node was described as being responsible for controlling the resource allocation and communications within a WiMAX cell, as described in the IEEE 802.16 standards. The WiMAX base station is also responsible for providing network services through the service flow functionality. The L3NP functionality described in Section 4.7 is an example of one of these services. Once the service flow is established, the BS node is responsible for transmitting traffic between the SS nodes and the service provider. In the case of the L3NP, the IAG is the service provider node.

Communication between the BS node and the IAG node occurs over the WiMAX service provider's backend network using the WBNP described in Section 4.8. Upon initial system startup, the BS node establishes a connection with the IAG node using the WBNP. Once this network link is established, the BS node will transmit all L3NP packets received along L3NP service flows to the IAG node using this link.

When the BS node receives a valid packet from an SS node identified with the L3NP CID, the payload of the message will be bundled into a WBNP packet, along with the CID, and sent to the IAG node using the WBNP. Upon receipt of a WBNP packet from the IAG node, the BS node will examine the CID and determine if it is a valid SF-CID. If so, the BS node will transmit the packet to the node using that CID across the WiMAX/DSRC network. Across the WiMAX network, this traffic is handled like any other traffic. This functionality is explained in detail in Section 4.8.5.

4.5 Network Server (NS) Module Functionality

The Network Server module operates as an application within the IAG node. Using the underlying WBNP connections provided by the underlying IAG module, it functions as a server for L3NP sessions within the system. Using the protocol described in Section 4.7, end users in the system can open a connection to the Internet using the L3NP. The NS module is responsible for providing the server functionality described in the L3NP protocol.

The ultimate goal of the IAG node is to provide Internet access for users in the system. The L3NP provides a PPP-like service for establishing an Internet connection, using the IPCP protocol for actually establishing an Internet protocol connection over the L3NP system.

As with a PPP server, the NS is only responsible for establishing and coordinating the network layer connections transmitted over L3NP. The actual duties of routing traffic to and from the Internet, as well as establishment of the connections, assignment of IP addresses, etc, is the responsibility of another component of the system. This functionality is outside of the research covered in this thesis, but is assumed to take the form of a router controlling the subnet of IP addresses assigned to WiMAX users. User traffic is sent to from the NS to the router in the form of raw IP packets. Traffic received from the router is also IP traffic. The destination address in the IP header is checked in the NS routing table to find the session assigned to that address.

4.6 Access Concentrator (AC) Module Functionality

The Access Concentrator module is an application that operates on the subscriber station node. It functions as an L3NP client, connecting to the NS module over the underlying WiMAX/DSRC network. The AC module is responsible for implementing the L3NP client functionality described in Section 4.7.

The overall purpose of the AC module is to provide Internet access for end user Internet applications hosted on the SS node. The AC module is responsible for collecting all the traffic from this user traffic, encapsulating it in L3NP packets and sending it across the WiMAX/DSRC network (to be processed by the NS module in the IAG node). Likewise, traffic for these user applications are received at the AC module in the form of encapsulated frames. These have to be transmitted to the user applications. The specifics of the interface between the AC module and the end user applications/TCP stack are left to the specific implementation.

4.7 Layer 3 Network Protocol (L3NP) Functionality

The goal of system described in this thesis is to provide an end-to-end solution for robust Internet access in vehicular platforms. The combined WiMAX/DSRC network, described in the previous chapter, provides a link layer connection that maximizes bandwidth through a minimization of overhead and provides robust support for mobility.

The Layer 3 Network Protocol (L3NP) completes the system by providing network layer access to the Internet across the WiMAX/DSRC network. The L3NP, in conjunction with the WBNP described in Section 4.8, add the support for Internet access across the system that is rapidly updated as nodes migrate through the system, without requiring a change in IP address or reestablishment of Internet connections.

4.7.1 System Structure

The L3NP protocol is based heavily on the Point-to-Point Protocol (PPP). As described in Chapter 1, PPP is a network-layer protocol that operates over top of an underlying link layer, providing a framework to provide network access while minimizing the addition of overhead. PPP is a popular protocol and has been extended several times beyond its initial applications.

The software modules used to implement L3NP are the Access Concentrator (AC) module described in Section 4.6 and the Network Server (NS) module described in Section 4.5. The NS module operates as a process in the IAG node and provides the server functionality described in this chapter. Each end user (SS node) uses an AC module as a client process to access the NS module operating in the IAG node and open a connection to the Internet. The two nodes connect over the underlying WiMAX/DSRC system to establish an L3NP session.

4.7.2 L3NP Frame Structure

The packet structure for the L3NP is based on a modified version of the PPP packet. The Session ID (SID) field is added to identify the source client of a particular packet and to allow for easy local mobility without service interruption. This unique identifier is assigned to each client when they initially enter the system and is used for all future communications between the client and the server. Figure 4.2 shows the L3NP frame structure, based on this modified PPP frame structure.

LSB

Session ID (2-4 bytes – TBD)	**Protocol** (1 or 2 bytes)	**Data** (Payload dependant)

Figure 4.2: L3NP Frame Structure

The session ID field is a unique field assigned to each active session in the system. The protocol describes the contents expected in the data field. Table 4.1 lists several protocols used in the L3NP system.

Table 4.1: Protocol Field Values

Value (in hex)	Name
0x0021	Internet Protocol
0x8021	Internet Protocol Control Protocol
0xc021	Link Control Protocol
0xc023	Password Authentication Protocol
0xc223	Challenge Handshake Authentication Protocol
0xc227	PPP Extensible Authentication Protocol

Link Control Protocols

The Link Control Protocol is the primary control protocol used for PPP type connections. Defined by the PPP standard [**13**], this protocol is used to set up the connection between the client and the server and guide the system through authentication, network link establishment and link termination.

Authentication Control Protocols

The Password Authentication Protocol (PAP), the Challenge Handshake Authentication Protocol (CHAP) and the Extensible Authentication Protocol (EAP) are the common protocols used by the network to perform the authentication process. PAP is an older, plaintext based protocol while CHAP and EAP are newer and more secure protocols. The system will use one of these protocols to authenticate the user to the authentication server.

Internet Related Protocols

The primary focus of PPP in the WiMAX context is the Internet protocol. The Internet Protocol Control Protocol (IPCP) [**22**] is the network control protocol used to set up the IP session across the PPP link. Once the session is established, the IP traffic packets use the IP protocol field.

Session ID

The Point-to-Point Protocol was based on a constant connection along an established underlying link connection. In the event that the link layer connection is broken, the session was terminated and had to be re-established. A review into protocols derived from PPP found that none were developed to be robust against mobility in the link layer.

By attaching a unique ID to each session connected with the L3NP, the NS can maintain the connections associated with a session (particularly network layer connections), even when the underlying link connection is disturbed. The incoming traffic is linked to a particular session through the SID, which remains constant, even as the routing information is changed. The concept of routing in the L3NP system is discussed in Section 4.8.1.

4.7.3 Operation Phases

The phases of operation of the system borrowed heavily from the PPP standard. An additional handover phase was added, reflecting the mobile nature of the system. The state diagram of an L3NP session is shown in Figure 4.3.

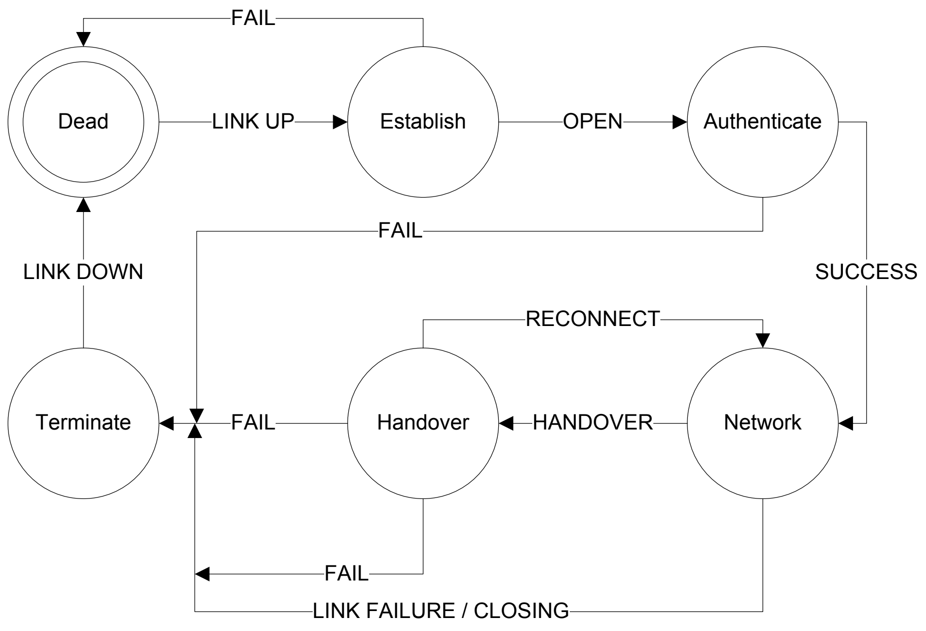

Figure 4.3: L3NP State Diagram

Link Dead (Physical Layer Not Ready)

This is the initial state of the system. No connection exists between the AC and NS nodes. The system is idling, waiting for the subscriber station to initiate a connection.

Link Establishment Phase

This stage establishes a connection between the AC and NS nodes. Using the underlying WiMAX connection for user traffic, link establishment consists of the AC and NS making their initial handshaking connection. This is conducted using the Link Control Protocol (LCP), described in the PPP specification.

The use of a Session ID is a significant change from the original PPP protocol. The user initially does not have an assigned SID and will instead use the default one for network entry (SID of 0) for its first LCP message. The network server will assign the user a SID at this point. This is included in the return message and is used for all future communications for this session.

An entry for the session is stored in an internal session table within the IAG node, consisting of the SID and the current status of the connection. Additional information, such as routing tables and assigned IP information, is handled by other components of the IAG node, with the session SID being used as the primary identifier.

Authentication Phase

The client must authenticate itself before being granted access to the network layer protocols. This step is modeled after the authentication used in PPP. An authentication protocol such as PAP, CHAP or EAP is used to connect with an authentication server to control the authentication, authorization and accounting (AAA) functions. In the interest of network security (as WiMAX is conducted over a broadcast air interface), this would also be the point where encryption set-up for the link could occur. Unlike L2TP, IPsec can’t be used due to a lack of underlying IP network. Instead, a public key handshaking can occur at this point, along with the establishment of session keys for symmetrical encryption (such as 3DES or AES). Analyzing security features across the combined WiMAX/DSRC network with L3NP will be the subject of future research.

Network-Layer Phase

The network phase is responsible for the actual network layer communication. Upon initial connection, the AC module will open up a new IP connection with the NS module using the Internet Protocol Control Protocol (IPCP). The NS module will provide network session information (primarily IP address), which allows the AC module to open up a network device with these parameters and update its routing table such that all user traffic will be routed across this network link. Once established, all Internet traffic will flow between the AC and the NS. From there, it will be routed to and from the Internet using the router functionality described in Section 4.5.

This phase is maintained through regular traffic or the periodic messages from idle subscriber station nodes. If this hasn’t been received from a node for some time, the node will automatically switch to the handover phase. The timer times are based on values discussed in Section 4.6 of the PPP standard. Routing updates in a Cellular IP system is discussed in more detail in [**12**]. The timer values used in this standard will have to be adjusted to better suit the highly mobility environment, making them long enough to allow the system to initiate full handovers (and recover from failed handovers) but short enough to prevent wasting of resources for dead connections. Furthermore, switching into the handover phase (described in the next section) should be done in a timely manner in order to limit the packets dropped and the WiMAX resources wasted with unroutable traffic. The definition of these values will require analysis using an actual WiMAX/DSRC test bed in an actual end user environment. This will be investigated in future work once this test equipment becomes available.

Handover Phase

The handover phase is a new addition over the standard PPP protocols in use today. The connection is in a handover phase when the underlying WiMAX protocol is in the middle of changing connections from one serving Relay Station or Base Station node to another. There will also be network routing changes when a mobile Relay Station node hands over from one base station node to another. This system assumes that cross-layer information is not available (the AC module doesn't know that it is about to hand over). As such, this phase is passively entered. The AC module is unaware of the WiMAX network change, as it is handled by the underlying system. The AC module will continue to transmit after the handover without any change.

The NS will become aware of handover in one of two ways. If the handover is delayed, the periodic location update will not be received in time and the timer in the network phase will expire. As well, any attempt to forward messages to the client will result in a routing failure. In the event of either of these occurring, the system will enter the handover phase and a handover timer will be started. When the node is in the handover stage, all forwarding of traffic from the server to the client is suspended. Depending on the implementation, the traffic can be temporarily cached until the node exits the handover state, at which point it will be forwarded along. Alternatively, the traffic is just dropped and the TCP/IP protocol is used to resend the traffic once the handover is complete.

In the event of a successful handover, the NS node will be aware of traffic being received from the same session (SID and any security/authentication matches), but the BSID and/or CID has changed. The IAG node will update its routing tables and the network layer phase will be continued. This passive update of the routing table follows the Cellular IP model of micromobility. The routing used in the system will be discussed in full in Section 4.8.1.

If an update is not received in a timely manner once the system enters the handover phase, the handover timer will expire. At this point, the system will assume that the connection is lost and the NS will move the connection into link termination. Much like the timers in the network phase, the exact values of the time-out timer for the handover phase will have to be the subject of future research based on an actual hardware platform and trials in the vehicle environment.

Link Termination Phase

The link termination phase occurs after failed authentication, a link failure or a failed handover attempt. It also takes place when the client communication is complete (user "hangs up"). Within the NS, all sessions will be closed and all resources (such as IP addresses) returned to the pool. The authentication server will be notified that the user is no longer connected. For the end user to access the Internet at this point, it will have to reestablish a link-layer connection to the server and go through the establishment phase once again.

4.8 WiMAX Backend Network Protocol (WBNP) Functionality

The WiMAX service provider's backend network is used for communications between Base Stations nodes (to negotiate handovers, for example) and to provide user services such as voice communications and Internet access. The establishment of such a network is required in order for the BS nodes to provide service. The Layer 3 Network Protocol described in Section 4.7 is an example of one of the services offered by WiMAX base stations and is provided through the backend network.

In order to offer the complete L3NP system, a protocol had to be developed to handle traffic between the Base Station nodes and the Internet Access Gateway node providing the L3NP service. In addition to transmitting the L3NP packets themselves, reverse path information had to be sent in order for the IAG node to properly route packets back to users.

As with the DSRC network, the implementation of this layer is somewhat dependant on the network structure used by a particular provider. For the sake of simplicity, a TCP/IP style network is once again assumed.

4.8.1 Routing

The major addition that L3NP, described in Section 4.7, makes to the Point-to-Point Protocol support for node mobility. The addition of Session IDs to the frame allows for the unique identification of a session, and thus a user, even as they migrate through the system. To make the system work the system also requires the reverse routing to the user. This allows the IAG node to route L3NP traffic to a specific user.

To establish a complete routing path, two pieces of information are used to identify a complete route to the user. The first is the Base Station ID of the BS node currently serving the user. The BSID uniquely defines a particular base station within the system. The second piece of information is the SF-CID currently assigned to the user by that BS node. The SF-CID is assigned during the network entry process described in Section 3.9.2 and is used specifically for L3NP traffic. This SF-CID is unique to an individual user within a WiMAX cell (controlled by a specific BS node) and can be used to uniquely identify the user. Through the combination of the BSID and the SF-CID, a unique reverse path can be assigned to a particular session. The BSID allows the packet to be routed to the proper base station, and the SF-CID is used by the BS node to route the traffic across the WiMAX network to a given user (with the exact routing across relay stations handled by the WiMAX network).

The BSID assigned to a particular BS node is determined during initial connection establishment, described in Section 4.8.4. The SF-CID for a particular user is transmitted as part of the packet structure described in Section 4.8.2. This information forms the basis of routing tables and is linked to a Session ID.

The network layer protocol was designed with mobility in mind. The WBNP was designed to support the principles of micromobility and allow movement through the system without interruption to the Internet connections. When the L3NP connection is initially established (and the session ID assigned), the system notes the BSID and CID the user is connecting from. This is stored in a routing table within the IAG node. Every time traffic is received from a user, the source BSID and CID is compared to the one in the routing table. If the user has moved in the system and has a different BSID and/or CID, these are updated in the routing table, keeping the table up-to-date. When the IAG node needs to transmit information to a specific session, it uses the cached BSID and CID values to properly route the message.

4.8.2 WBNP Frame Structure

The WBNP has a simple frame structure, seen in Figure 4.4. The possible message type values are listed in Table 4.2 and indicate how the data field should be interpreted. The length field indicates the total length of the packet in bytes. The CID field allows the IAG to update its routing tables (in the uplink direction), and indicates the routing for the packet (in the downlink direction). Finally, the data field contains the message data relevant to the message type.

LSB

Message Type (1-2 bytes – TBD)	Length (2 bytes)	User CID (2 bytes)	Data (Variable)

Figure 4.4: WBNP Frame Structure

4.8.3 Message Types

The message type field in the WBNP packet specifies the contents of the WBNP packet. The majority of the packets will be user data, with a message type of DATA. The other message types are for management purposes, in order to establish or terminate the connection and to report routing errors.

Table 4.2: WBNP Message Types

Message Type Code	Message Type Name	Description
1	DATA	Tunneled user connection
2	HS-REQ	Handshake Request
3	HS-ACK	Handshake Acknowledgement
4	TERM-REQ	Terminate Connection
5	ERR-NOROUTE	Error – Cannot route

DATA

This packet contains a L3NP frame, as described in Section 4.7.2. The data field is not interpreted, but forwarded to the upper layer NS and AC modules for processing as described in Section 4.7. DATA messages can only be sent once a WBNP connection is established between the BS and IAG nodes. The CID is set to the SF-CID assigned to the user.

HS-REQ

This packet is sent by the BS node to initiate the connection between the BS and IAG nodes. The data fields, as seen in Table 4.3, indicate the supported functionality of the node, as well as the BSID for the purposes of routing. The User CID field is ignored.

Table 4.3: HS-REQ Fields

Field Name	Description
Version	The highest protocol version supported by the BS node
BSID	The assigned BSID for the BS node

HS-ACK

The reply packet from the IAG node to the BS, seen in Table 4.4, indicates the protocol version used for this communication. This is the lowest of the BS and IAG supported versions. The User CID field is ignored.

Table 4.4: HS-ACK Fields

Field Name	Description
Version	The protocol version that will be used for the connection

TERM-REQ

This packet can be sent by either the BS or IAG node to terminate the connection between the two nodes. All NS sessions currently being served by this BS node will be suspended until a new route to each AC module is established or the session times out. The User CID field is ignored.

No fields are attached to this message.

ERR-NOROUTE

This packet is returned by a base station node to indicate that the CID used in a DATA message is no longer valid. This alerts the IAG node that the route for a given connection is no longer valid. The main purpose of this message is to indicate a recent handover of that client (as the CID has not yet been reassigned). The NS module will put that session into the handover phase, as described in Section 4.7.3. The User CID field is set to the invalid CID value.

No fields are attached to this message.

4.8.4 Network Connection Establishment

Initial network connection between the IAG node and a BS node across the backend network begins with a network connection. Once a TCP/IP connection between the two nodes is established, the BS node begins a two step handshake process.

1. The BS node sends the IAG node a HS-REQ message to request handshake and connection. The message includes the highest protocol version supported by the BS node, as well as the assigned BSID for that BS node.
2. If the handshake is successful, the IAG replies with a HS-RSP message. The included version number is the highest supported protocol version supported by both the BS and IAG nodes and the protocol that will be used for all communications.

At this point, the IAG and BS nodes are connected and prepared for communication. The protocol assumes that the backend network is robust and nodes will remain connected unless the BS nodes are brought down for maintenance or reconfiguration. The IAG node will keep a record of the BSID associated with this connection for routing purposes.

4.8.5 Normal Operation

Once the handshake process is complete, the system is ready for transmission of L3NP packets between the BS nodes and the IAG node. These packets support network layer traffic over top of the links established across the WiMAX/DSRC network and using the WBNP. This traffic is transmitted across the WBNP links using DATA messages, as described in Section 4.8.3.

BS to IAG Communications

L3NP data is received at the BS node through L3NP service flows established with SS nodes during the initial network entry stage. By using the SF-CID, the user indicates that the contents of the frame are destined for the IAG node (and ultimately the NS module).

Upon receiving a frame with the SF-CID as the source, the BS node prepares a WBNP frame. The payload of the message is copied into the data field and the SF-CID is put into the CID field. The message is then forwarded across the WBNP link to the IAG node.

Once received at the IAG node, the L3NP payload is retrieved from the WBNP frame. The session ID is read off of the L3NP frame and used in conjunction with the CID field and the BSID associated with that connection to update the routing tables. The L3NP payload is then sent to the NS module for processing.

IAG to BS Communications

When the NS module in the IAG node has an L3NP frame for a specific session, it looks up the BSID and CID associated with that Session ID in the routing tables. A WBNP frame is created using the L3NP frame as the payload and the CID value taken from the routing table. Looking up the destination BSID, the IAG node sends the WBNP frame across the WBNP connection associated with that BS node.

When the message is received at the BS node, it will first check to see if the CID is valid. If so, the BS node will encapsulate the attached user data into a service flow message for the SS node with that CID. This will be transmitted across the WiMAX/DSRC network like all other traffic. If the CID is invalid (for example, the user has recently handed over to a new node), the BS node sends a WBNP message back to the IAG node with an ERR-NOROUTE message type and the invalid CID, indicating routing failure.

4.9 Summary

This chapter built on the combined WiMAX/DSRC link layer described in Chapter 3. To provide a complete vehicle based mobile solution, a network layer protocol was built on top of the underlying link layer. Based on the Point-to-Point Protocol, the Layer 3 Network Protocol allows end users to access across the Internet across the WiMAX/DSRC system.

Based on the system described in Chapters 3 and 4, a demonstration system was constructed to explore the feasibility of the proposed system. This partial implementation demonstrated the functionality of the various components of the system, as well as the communication between the components. The demonstration system only models the functionality of the MAC layer and above. The details of the demonstration system are described in Chapter 5.

Chapter 5: Demonstration System

5.1 Introduction

The previous chapters in this thesis have described an Internet access system for vehicle platforms. A combined WiMAX/DSRC link layer was described in Chapter 3. This system improved the efficiency of the existing Mobile WiMAX protocol through the use of DSRC clusters. The DSRC cluster heads also functioned as 16j WiMAX relay stations and concentrated the WiMAX connections of all the cluster members. On top of this link layer connection, Chapter 4 added a network layer protocol called Layer 3 Network Protocol. Based heavily on the Point-to-Point Protocol, L3NP allows an end user node to establish a network layer connection to the Internet overtop of the combined WiMAX/DSRC infrastructure. PPP was extended to provide support for the rapid level of mobility experienced by vehicle-based platforms. By adding session IDs and routing based on Cellular IP, nodes are able to maintain their Internet sessions throughout their migration through the WiMAX system.

This chapter details the design of a demonstration system created to test the concepts described in the previous chapters. The demonstration system implements both the combined WiMAX/DSRC system, as well as the L3NP and WBNP protocols. While not a complete implementation of the proposed system, the demonstration system will function as a core component of a future research platform. All the code used to develop the demonstration system is included in Appendix B.

5.2 System Design

The demonstration system implements part of the entire system described in Chapter 3 and 4. The main goal developing a demonstration system was to demonstrate that the system structure was sound, and to produce a system framework to support future research work. This section discusses the development environment used to produce the demonstration system and the limitations in the scope of the system.

5.2.1 Development Environment

Primary development of the demonstration system was completed in the C# language using Microsoft Visual Studio 2008 and the Microsoft .NET framework. C# is a modern, object-oriented, strongly typecast language with a complete base class library of functions. Like many modern object oriented languages, C# avoids the use of pointers and has robust exception handling. C# and .NET has full support for multithreading, which was used extensively for the implementation of the system. The development process was also greatly simplified using the Microsoft Visual Studio development environment, which includes a robust debugger, syntax highlighting and extremely complete automatic code completion for the C# language.

The Microsoft .NET framework operates by compiling all languages into a generic binary, or bytecode, called the Common Intermediate Language (CIL), as part of the Common Language Infrastructure (CLI). This binary is independent of platform and allows all the programming languages supported by .NET (such as C#, C++ and Visual Basic) to access both the same base class functions, and to allow interoperability between code written in different languages. To execute the code, a just-in-time (JIT) compiler translates the intermediate bytecode into native binary commands for the specific platform. This is similar to the virtual machine concept used in the Java programming language. The Common Language Runtime (CLR) is Microsoft's implementation of the .NET JIT compiler. At the present time, the .NET framework is at version 3.5.

Software produced under the .NET framework is not restricted to Microsoft-based platforms, however. The Mono project is an open-source implementation of the .NET CLR based on the same Emca standards defining C# [**23**] and the CIL [**24**]. It allows for .NET binaries to be operated in other operating systems (including Linux and Apple OS X) and non-x86 platforms (including SPARC and ARM). Mono implements the JIT compiler and the base class library. At the present time, Mono supports the features of the .NET framework up to version 2.0, with support for the 3.0/3.5 features currently under development. The demonstration system can be operated on any platform with a Mono runtime, and was demonstrated on a Linux-based platform.

The final important component of C# allows for access to shared libraries. The C# language and the .NET framework in general were designed to keep hardware abstract. However, sometimes access to external libraries is required. C# includes functionality called Platform Invoke (P/Invoke) to access functions in linked libraries. Under Windows, linked libraries are Dynamic-Link Library (DLL) files. Mono has extended this to other styles of linked libraries, and this is used in the demonstration system.

The demonstration system requires access to a Linux kernel module in order to provide the required functionality, and C# was unsuitable for this functionality. A small module was written in the C language and compiled as a linked library using the GCC compiler to handle the interface between the C# program and the low level commands required to use this kernel module. The P/Invoke functionality of C# was then used to allow the main program to access the kernel module through the shared library. This will be described in Section 5.6.

5.2.2 System Scope

The demonstration system was created to demonstrate the feasibility of the system described in Chapter 3 and 4. It was also produced as the framework for a test platform to conduct future work on the proposed system. Due to a number of factors, the demonstration system did not implement all the features in Chapter 3 and 4, and will leave some functionality for future research work.

At the time of development, a physical test bed with DSRC and WiMAX radios was not yet available. As the work in Chapter 3 and 4 focuses primarily on the MAC layer and above, the demonstration system focused on these aspects of the complete system. The physical layer interfaces, including DSRC and WiMAX, was simulated using TCP/IP connections across a local area network. The system is designed to be the core of a research system based on actual DSRC and WiMAX radios once the equipment becomes available, at which point more detailed studies on system performance can be undertaken and the system can be further optimized.

One major component of the system described in Chapter 3 that was not implemented was the handover procedures. The demonstration system focused on the basic functionality of the system, including the establishment of connections between nodes and the initial network entry processes described in Sections 3.9, 4.7 and 4.8.4. As well, the normal operation of nodes was also demonstrated, showing proper operation of both the WiMAX/DSRC link layer and the L3NP/WBNP network layer. The handover processes will be implemented in future work. Real-world testing of handover scenarios will allow the performance of the handover process to be measured and optimized.

Finally, as mentioned in Section 4.5, the implementation of the Internet routing functionality is highly implementation-specific and was not part of the system being developed in this thesis. The development of this functionality could be part of future work in an attempt to optimize the system performance metrics once a test bed is available. For the demonstration system, all IP traffic received by the NS module will be repackaged and sent back unchanged to the source node. This will allow the system to demonstrate full end-to-end functionality of the L3NP. This will be described in more detail in Section 5.4.6.

5.3 System Structure

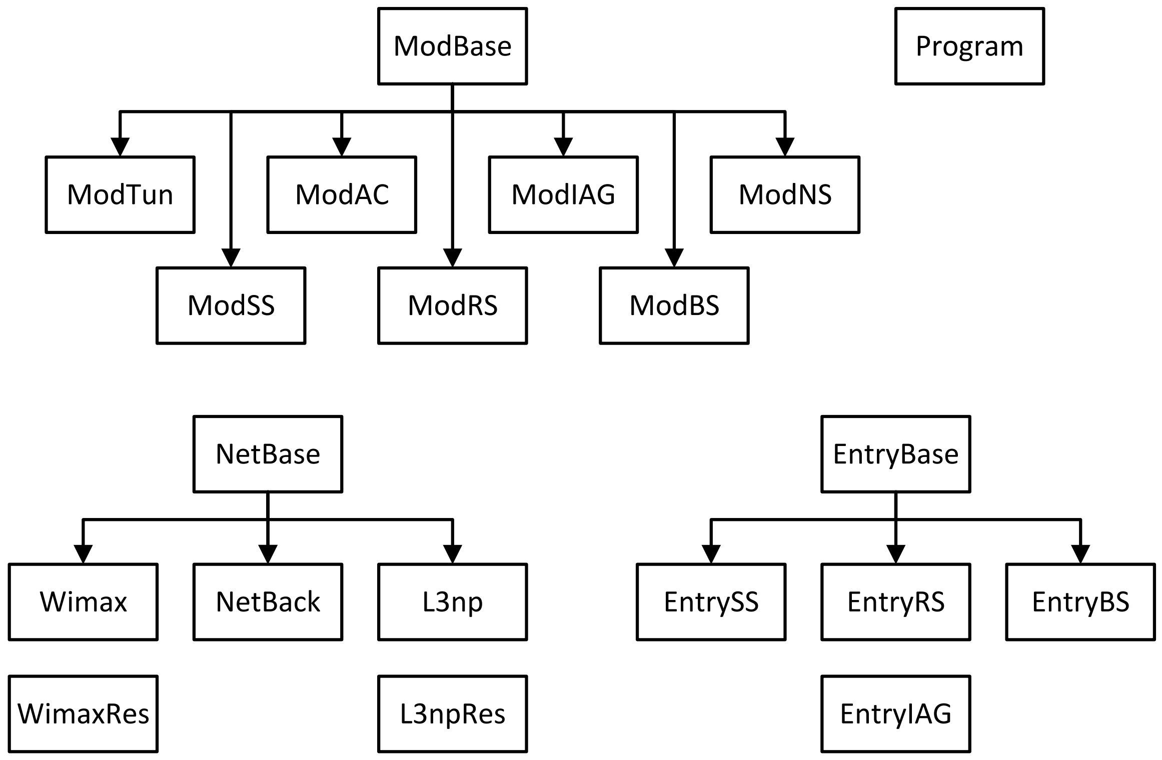

Figure 5.1 shows a class diagram of the demonstration system. While made up of many components, the classes break down into four major types: System modules and network layer applications (inherited from ModBase), networking protocols (inherited from NetBase) and system node entries (inherited from EntryBase).

System Modules

The system modules are the implementation of the major nodes described in Chapters 3 and 4. These include the Subscriber Station (SS), Relay Station (RS), Base Station (BS) and Internet Access Gateway (IAG) nodes.

In the demonstration system, all four major modules (SS, RS, BS and IAG) are being implemented with the same software, as opposed to four different programs. On program start-up, the user selects which node it wishes to run (and optionally, change system parameters). There is a lot of code overlap between the different nodes and it makes sense to take this approach for a demonstration system. The code for these modules is included in Appendix B.3.

Network Layer Applications

The Access Concentrator (AC) and Network Server (NS) are network layer modules that operate on the SS and IAG nodes, respectively. They are responsible for establishing the L3NP protocol connection, described in Section 4.7, over top of the underlying WiMAX/DSRC connection. They operate as a separate thread within the service module process.

There is a final module, called the Tunnel module (ModTun), which is related to the network layer applications. As described in Section 4.7, the AC module is responsible for collecting traffic from user applications and delivering IP traffic received over the L3NP connection back to those applications. This module, which interfaces with the AC module and also operates as a separate thread within the SS node, provides this functionality. The Tunnel module will be detailed in Section 5.6. The code for these modules is included in Appendix B.4.

Network Protocols

Networking protocols are used to implement the communications between modules. The three protocols implemented within the system are WiMAX (Wimax), the WiMAX Backend Network Protocol (NetBack) and the Layer 3 Network Protocol (L3np). These classes define the frame structures used in each protocol, provide the mechanisms for preparing frames for sending and decoding incoming frames and provide the message handlers to allow a node to properly respond to different kinds of incoming message types. The source code for these modules is included in Appendices B.6, B.7 and B.8.

System Node Entries

The node entries were developed to store status information about the four system nodes (SS, RS, BS, IAG). They contain information for a particular node, such as assigned CIDs/BSIDs and the socket information for network connections. In addition to the current node, these entries are also used for other nodes that the current node is connected to, with tables of these entries forming the routing tables.

For example, a RS node is connected to an upstream BS node across WiMAX and to multiple SS nodes across DSRC. The RS node retains entries for itself (ModRS), for its serving BS node (ModBS) and for all the SS nodes it is connected to (ModSS). As it is connected to multiple SS nodes, it stores this information in a linked list, which can later be queried to route a traffic message. The code for these modules are included in Appendix B.9.

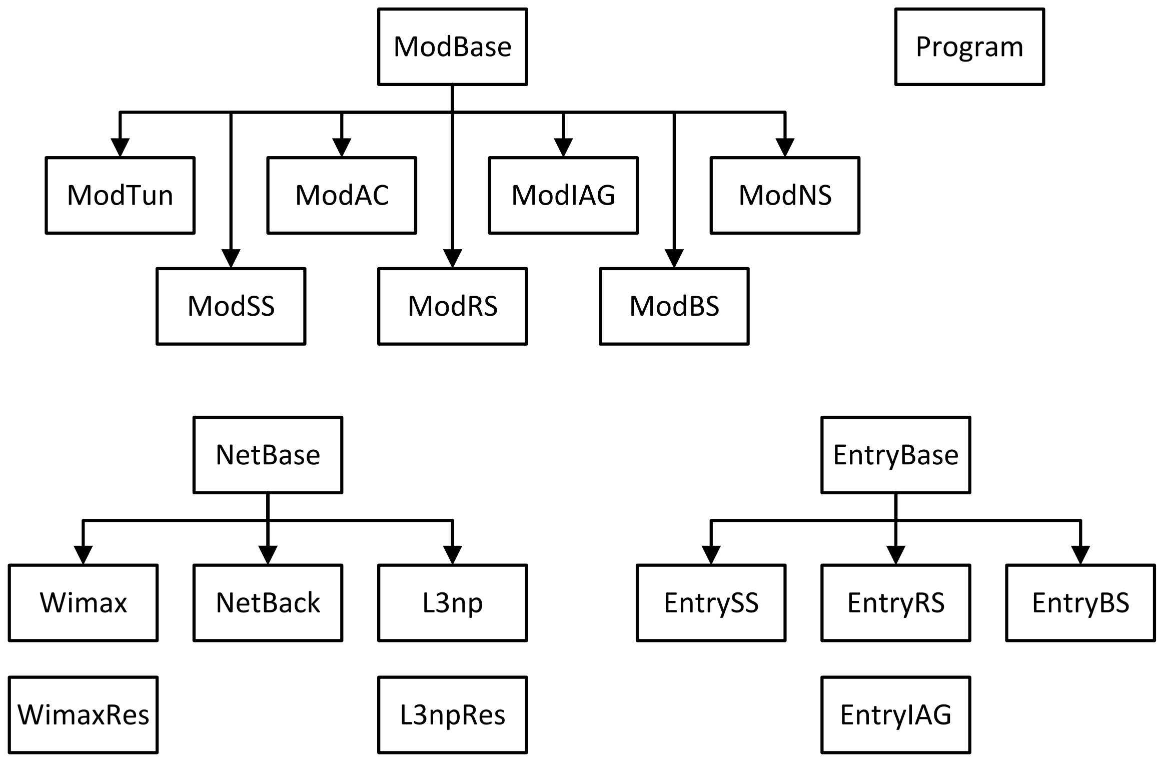

Figure 5.1: Demonstration System Class Diagram

5.4 Module Descriptions

The demonstration system was broken into four system node types, reflecting the four types of nodes present in the proposed system described in Chapters 3 and 4. This includes the SS, RS, BS and IAG nodes, which are responsible for the link-layer communications links. The AC and NS modules handle the network layer communications between user applications and the Internet using the L3NP protocol.

All communications between modules followed the C# model of asynchronous communications, with separate threads handling incoming and outgoing messages. The sending thread has a send queue for sending, which is implemented as a C# queue and is periodically queued. If there is a message waiting to be sent, it is sent across the connection using the underlying link send command.

The receiving thread blocks until a message is received. A handler is called to process the received message. In a manner described by the specific module, the handler decodes the message, performs any necessary processing, send a reply message (if required), and then blocks for the next message. This is usually handled by the message handlers defined within each protocol.

Every connection is formed in this manner and has its own set of threads. Using an RS node as an example, it has two threads for the connection between the RS node and the BS node, as well as two threads for each connection between the RS node and each SS node connected to the WiMAX system through the RS node. TCP/IP connections are used to represent the DSRC, WiMAX and WiMAX service provider backend connections.

5.4.1 SS Node (ModSS)

As described in Chapter 3, the Subscriber Station (SS) node represents the end user in the system. Within the context of the proposed system, it is a vehicle node connected to a Relay Station (RS) node across a DSRC network. The entire SS node structure as it was implemented in the demonstration system, including higher level modules, can be seen in Figure 5.2.

The SS has one bi-directional network link, operating between the BS node and the serving RS node. The demonstration system uses a TCP/IP link to simulate the DSRC link. It uses the WiMAX protocol described in Section 5.5.1 for these communications. The SS node establishes this connection during initial network entry and then performs the network entry process described in Section 3.9.2.

The SS module is also connected to the AC module (ModAC). As described in the next section, the AC module is responsible for managing network-layer connections. A queue in ModSS is used to store L3NP frames from ModAC, which are subsequently forwarded across the WiMAX network. Likewise, L3NP frames received from the RS node are then forwarded to a queue within the AC module.

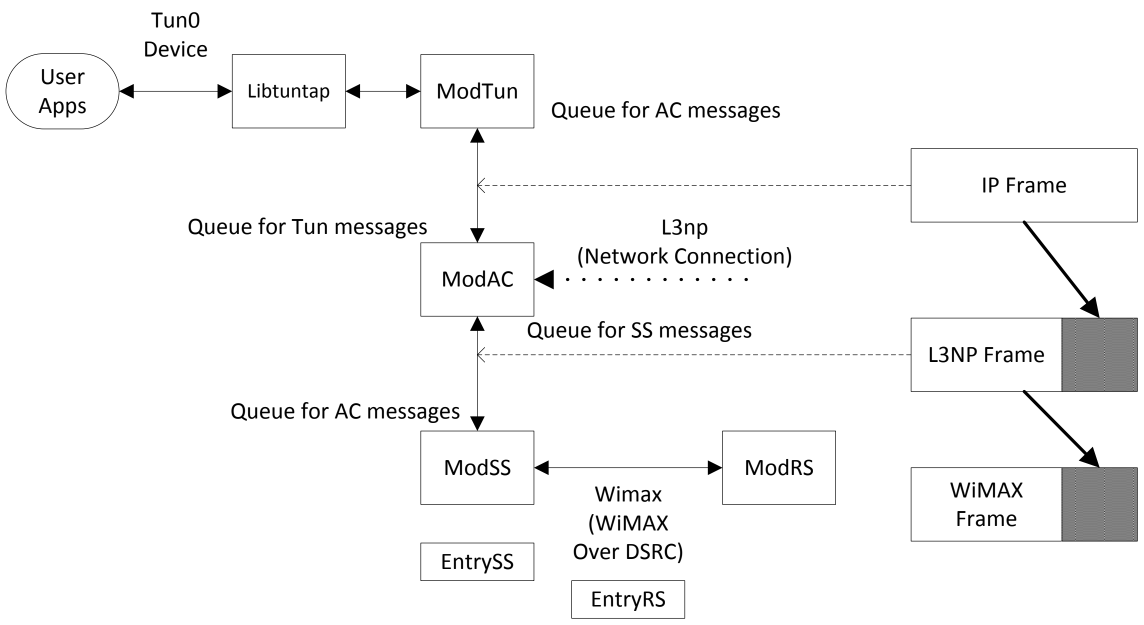

Figure 5.2: Subscriber Station Node Structure

5.4.2 AC Module (ModAC)

The Access Aoncentrator (AC) module operates as an application-layer subprocess within the SS node. It is spawned as a separate thread by ModSS during system start-up. It is responsible for implementing the client functions of the Layer 3 Network Protocol as described in Section 4.6.

Communications were established between the AC module and the Network Server (NS) module operating in the Internet Access Gateway (IAG) node using the L3NP protocol described in Section 5.5.2.

The demonstration system assumes that the user applications will be operating on the same system as the SS node functionality. This design decision influences how the user application traffic is captured by the AC module. The AC module communicates with the user applications through the ModTun module, which will be explained in detail in Section 5.6. A queue within ModAC is used to store IP frames received from the Tunnel module, which are encapsulated into an L3NP frames and set along to the IAG node using the underlying ModSS. Similarly, IP frames which are received by the ModAC within L3NP frames are forwarded to a queue within the Tunnel module.

5.4.3 RS Node (ModRS)

The Relay Station node functions as both the cluster head in the DSRC network and a 16j relay station node within the WiMAX network, as described in Section 3.5. It is responsible for transmitting WiMAX traffic between the nodes served by it on the DSRC network and the base station on the WiMAX network using the WiMAX relay messages described in Section 3.7.4. The system structure can be seen in Figure 5.3.

The RS node has two classes of connections. It has an uplink connection to the BS node, conducted over a simulated WiMAX connection. It also has multiple connections to SS nodes across the simulated WiMAX-over-DSRC connection, where it functions as a server. The connection to the BS node is established during initial system entry and performs the process described in Section 3.9.1. This connection is through the simulated WiMAX system and using the WiMAX protocol described in Section 5.5.1. The system also opens up a TCP/IP server to accept connections from SS nodes. Connections received by the server spawns a new thread for each connection. Each connection also creates an SS entry, which is stored in a linked list and forms the basis of the routing table. The details in the SS entries are updated as the SS node goes through the network entry process, caching copies of details such as the CIDs assigned to the user and its current authentication status.

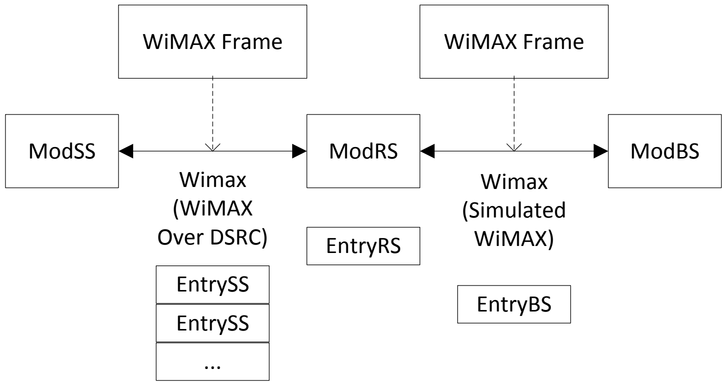

Figure 5.3: Relay Station Node Structure

5.4.4 BS Node (ModBS)

The WiMAX Base Station node functions as a 16j-enabled WiMAX base station. It is responsible for controlling system resource allocation and controlling all traffic going through the WiMAX cell it controls. ModBS provides the functionality described in Section 3.6. The layout of the BS node can be seen in Figure 5.4.

On initial network entry, the BS node establishes a connection across the simulated WiMAX service provider's backend network to the IAG node using the WiMAX Backend Network Protocol, described in Section 5.5.3. A connection is established using the process described in Section 4.8.4.

The BS node also opens up a TCP/IP server to accept connections from RS nodes across the simulated WiMAX cell. When RS nodes connect, a new thread is spawned for each connection and an RS entry is added to a table of connections to form the routing table. Using the multihop WiMAX standard described in the 16j amendment, the BS node is also indirectly connected with the SS nodes served by the connected RS nodes. Traffic between the BS node and the SS nodes is sent through tunnel connections with the RS nodes. As the BS node is also responsible for allocation of resources for the SS nodes and routing their traffic, it also maintains a table of SS entries. These SS entries include the serving RS station, allowing the BS node to properly route traffic.

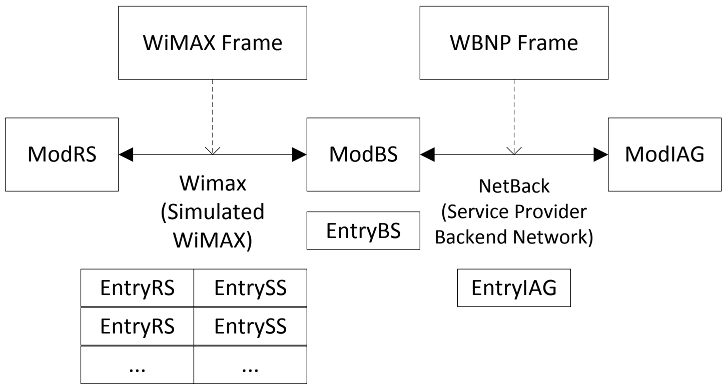

Figure 5.4: Base Station Node Structure

5.4.5 IAG Node (ModIAG)

The Internet Access Gateway node is responsible for providing the L3NP network-layer access service to BS nodes throughout the system, allowing SS nodes to establish a L3NP connection and gain access to the Internet. ModIAG provides the functionality described in Section 4.3. The entire IAG node structure as it was implemented in the demonstration system, including higher level modules, can be seen in Figure 5.5.

The IAG node opens a TCP/IP socket to accept connections from BS nodes across the WiMAX service provider’s backend network. Every time a BS node connects, a new thread is spawned to manage the connections with that BS node. The IAG node functions as a server for the WBNP protocol.

The IAG module is internally connected to the network server module (ModNS), which processes the L3NP frames that are received across the WBNP connections. The Interop Frame shown in Figure 5.5 is an internal frame structure used for communications between the main IAG module and the NS module. As discussed in Section 4.8.1, the IAG and NS modules use a combination of BSID and CID for routing L3NP traffic to a specific user. The Interop Frame contains the source or destination BSID/CID pair, as well as an L3NP frame. WBNP frames received by the IAG node are reassembled into Interop Frame structures and added to a queue in the NS module. Likewise, the IAG module contains a queue that accepts Interop Frame structures from the NS module, which are then reassembled into WBNP frames and sent to the appropriate BS node.

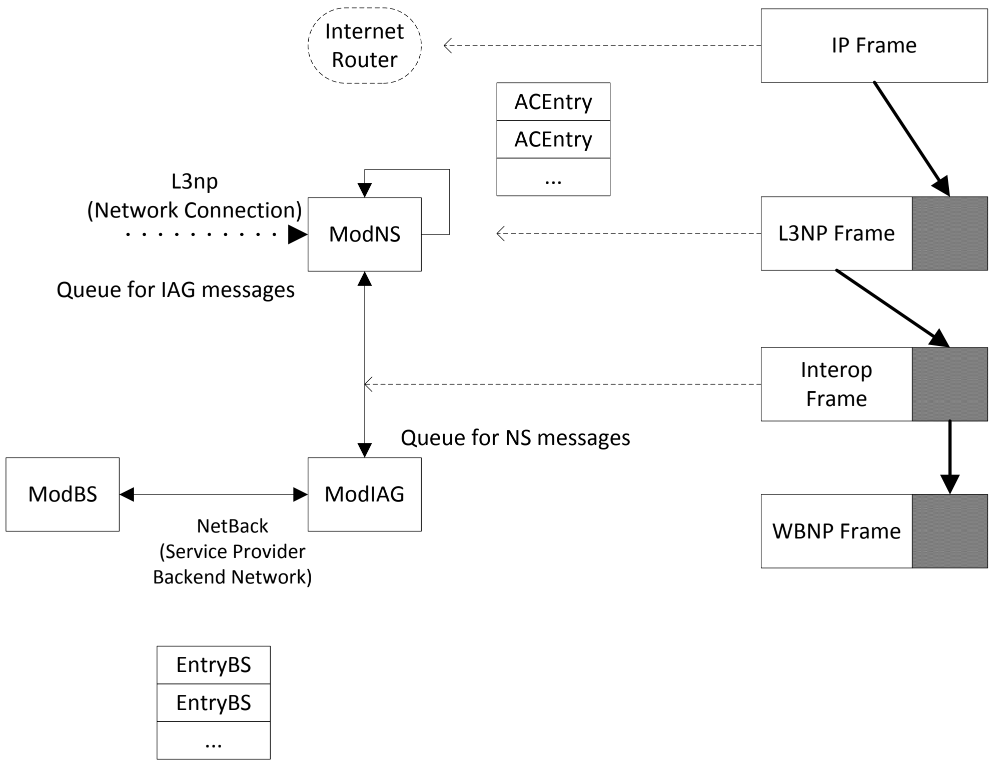

Figure 5.5: Internet Access Gateway Node Structure

5.4.6 NS Module (ModNS)

Much like the AC module, the Network Server (NS) module operates as an application-layer subprocess within the IAG node. It is spawned as a separate thread by ModIAG during system start-up. The NS module is responsible for implementing the server functions of the Layer 3 Network Protocol, as described in Section 4.5.

The NS module maintains a table of active L3NP sessions in the system through a linked list of ACEntry structures. This table stores the Session ID and status of sessions in the system, as well as the last reported reverse route (BSID/CID pairs) and assigned IP addresses.

As described in the previous section, the NS module receives traffic from the IAG module in the form of Interop Frame structures sent to an internal queue. L3NP frames are sent for transmission by the underlying IAG node by packaging the L3NP frame into an Interop Frame structure along with the destination CID and BSID from the ACEntry table.

Finally, as discussed in detail earlier in this chapter, the Internet Router functionality is left for future research. For the purposes of demonstration, any IP packets received at the NS module are not forwarded on. Instead, they are packaged back into an L3NP frame and sent back to their source. While not providing Internet access, this nevertheless demonstrates the end-to-end connectivity of the system and the functionality of the L3NP.

5.5 Network Protocol Descriptions

The network protocol classes implement the protocols necessary for communications between nodes. As the demonstration system doesn't yet have access to physical WiMAX or DSRC hardware, the links using these nodes are simulated using TCP/IP connections. However, all traffic in the system use the frame structure that is intended for actual implementation of the respective link.

As the system is relying on TCP/IP connections, the base class NetBase includes the underlying functions to establish TCP/IP connections, as well as the base functions for sending and receiving data. The network protocol classes build on top of these for their sending and receiving routines.

The protocol classes are constructed with a number of components. One major component is the definition of frame structures. This allows other modules to properly construct a frame. Functions are included to convert the frame structure (made up of the applicable fields) into a stream of bytes ready for transmission across the physical interface. Likewise, there are functions to convert a received protocol frame back into the structure for interpretation.

The protocol classes also include the handlers for the various classes of messages that are received. This allows the module to properly react to the message, depending on the class (server/client) of the node. This includes any required processing and the preparation and sending of a reply message.

5.5.1 WiMAX (Wimax)

The Wimax class implements WiMAX protocol communications. In the demonstration system, this includes both the Virtual WiMAX communication across between the SS nodes and the RS node across the DSRC interface (described in Section 3.8) and the WiMAX protocol communication between the RS nodes and the BS node (described in Section 3.7). The Wimax module implements all aspects of the WiMAX protocol required for the functionality covered by the demonstration system. This includes the initial network entry and normal system operation for the SS-RS link and the RS-BS link.

For the link between the RS and the BS, the network tunnel functionality is also provided. As specified in the 16j draft, all communications between the SS nodes and the BS node that pass through an RS node are tunnelled through a single connection.

5.5.2 Layer 3 Network Protocol (L3np)

The L3np class implements the L3NP communications between the AC and NS modules. This includes all the client and server functionality described in Section 4.7. The nodes using L3NP do not connect directly, so L3np doesn't implement the TCP/IP connection functionality. Instead, messages are converted into a stream of bytes and sent to a lower layer for transmission.

5.5.3 WiMAX Backend Network Protocol (NetBack)

The Netback class implements the WBNP for communications between BS and IAG nodes. It includes all the client and server functionality described in Section 4.8.

5.6 Tunnel Connection

The tunnel module (ModTun) was included in the system to complete the functionality of the AC module present in the SS node. As described in Section 4.6, the AC module is responsible for collecting network traffic from the end user applications and transmitting it across the L3NP network connection. Likewise, all Internet traffic received at the AC module needs to be transmitted to the user applications.

The tunnel module works in conjunction with a small shared library, libtuntap, in order to provide this functionality. The shared library is designed for a Linux platform and is used to access a kernel module that provides a virtual network device.

5.6.1 Description of the Universal TUN/TAP Driver

To make the system operate, it must intercept traffic to and from all user applications. One popular method of doing this is through the virtual network device. To the end user applications, it appears the same as any other network device (such as an Ethernet or WiFi card). However, instead of traffic being sent to a physical device, it is instead forwarded to application software. The software then provides some processing and sends the traffic along an actual network device. To intercept the user application traffic, the IP routing tables in the system are modified so that all traffic is routed by default across the virtual network interface.

Using a virtual network device is a popular approach to providing a tunneled link or network layer connection over top of an existing network connection. Popular examples using this approach include virtual private network (VPN) client software and other secure tunneling software such as OpenSSH.

One popular method of creating a virtual network device in Linux that is controllable through application software is the Universal TUN/TAP driver [**25**]. This kernel module has been included as part of the Linux kernel since the 2.4 kernel version, and is also available for other platforms, including BSD, Apple OS X and Windows. TUN/TAP is opened like an ordinary file or device under Linux. After configuration using IOCTL, the device is read from and written to like any other stream device in the system. The driver operates in one of two modes. In TUN mode, it operates as a virtual network-layer device. All traffic in and out of the device is in the form of IP frames. In TAP mode, it operates as a virtual link-layer device, with all traffic being in the form of Ethernet frames.

5.6.2 Code Structure

Accessing the TUN/TAP driver directly from C#/.NET was not feasible due to abstraction of the hardware layer present in .NET. Opening and configuring the driver requires a number of lower level, platform-specific system calls. To achieve this, a small interface library was written in C to handle the set-up of the virtual network device and then the transmission to and from the device. The Tunnel module (ModTun) is responsible for the interface between the C#/.NET program and the shared library handling the low level functions.

The functions TunOpen (open and configure the TUN/TAP driver), TunClose (close the virtual network device), TunRead (read IP traffic from the virtual network device) and TunWrite (send data to the virtual network device) are handled a shared library (libtuntap.so) compiled from C code. As discussed in Section 5.2.2, C# includes the P/Invoke functionality for accessing functions in shared libraries. The Tunnel module uses P/Invoke to gain access to these four functions in the shared library in order to set up and communicate with the virtual network device. The TUN/TAP driver is operated in TUN mode and all traffic is IP frames. The code for the shared library is included in Appendix B.1.

The Tunnel module polls the device for traffic. When traffic is received, it is added to a queue located in the AC module. Another queue in the Tunnel module also captures IP traffic from the AC module. This traffic is forwarded along to the network device.

Assistance in designing this interface came from Dr. Renato Figueiredo and the ACIS lab at the University of Florida. Their IPOP project [**26**] is also written in C#, operated under Linux using Mono and makes use of a thin shared library written in C used to interface with the TUN/TAP driver. Examining the code in the IPOP project helped with writing and compiling the shared library, as well as properly configuring the P/Invoke interface.

5.7 Summary

This chapter described a demonstration system created to test out the system for vehicular networking described in Chapter 3 and 4. Within the scope limitations laid out in the early parts of the chapter, the demonstration system provided the core components of a test system to simulate the initial network entry and steady state operation of the proposed WiMAX/DSRC system, as well as the L3NP/WBNP network-layer protocols.

The chapter began with a description of the development environment used for the demonstration system and the scope of the demonstration system. The structure of the C# program was shown, followed by a detailed description of how each system node was implemented. The structure of the communication protocols was also discussed. The chapter ended with a section describing the Universal TUN/TAP driver and how it was used within the demonstration system to provide full subscriber station node functionality.

Chapter 6: Conclusions and Future Work

6.1 Conclusions

This thesis set out to design a wireless Internet access scheme suitable for vehicle-based systems. The system was designed to provide broadband levels of bandwidth and controlled latency for future in-vehicle applications, such as streaming audio, video and Voice over IP (VoIP). The overhead added by the proposed system was minimized. Finally, on account of the highly transient nature of vehicle nodes, the system was designed to provide robust mobility support. It was designed to respond to handover events rapidly and without interrupting the Internet connection, which is essential for supporting future Internet-based applications.

The main contributions presented in this thesis began in Chapter 2 with a full analysis into the Mobile WiMAX system efficiency under different levels of congestion. This was based on previous work done for Fixed WiMAX systems, but updated for the S-OFDMA frame structure used in Mobile WiMAX. The model was tested in MATLAB and demonstrated a similar system performance issue, with system efficiency dropping below 60% with 70 active bi-directional connections. To address this shortcoming, a novel system structure was proposed that used DSRC to group nodes together and uses the tunnelling functionality of relay nodes in the 16j amendment to reduce the number of connections (and thus overhead) in the WiMAX system while still servicing the same number of users. This was the first time that DSRC was used to provide access to a WiMAX network, and that the 16j relay station functionality was extended to relay traffic across a non-WiMAX technology. Initial modelling showed a potential system efficiency improvement of over 30% in the most congested systems.

Chapters 3 and 4 described in detail the proposed system structure. Chapter 3 detailed the modifications required to WiMAX to allow it to operate over a DSRC connection (assumed to be a TCP/IP link). It outlined the process of initial network entry for nodes connecting to the WiMAX network across the DSRC network, as well as the functionality of the relay station node responsible for translating between the two network technologies. Chapter 4 outlined a network layer protocol used to provide Internet access over the WiMAX/DSRC network. The Layer 3 Network Protocol was based on the PPP protocol, with modifications to support the high level of mobility present in vehicle-based systems. The protocol was complimented with the WiMAX

Backend Network Protocol, which used to transmit data between the WiMAX base stations and the Internet Access Gateway and provides support for micromobility.

Finally, Chapter 5 detailed a demonstration system produced to both demonstrate the system structure outlined in Chapter 3 and 4, as well as providing the basis of a future test platform. This demonstration system provided most of the basis functionality of the system and proved that a combination of both the WiMAX/DSRC link layer and the network layer protocol could be used to provide Internet access to vehicle-based platforms.

6.2 Future Work

The work presented in this thesis is the first step towards the use of combined WiMAX/DSRC nodes in the field. There are three major work areas envisioned to continue the work described in this thesis towards completion. The first is the creation of a complete test system with proper radio interfaces. This will allow for real-world testing of the performance of the system. Computer modeling of the system will also provide invaluable system performance information. Using both of these together, the system benefits can be proven and the system can be further optimized. Finally, a security analysis on the complete demonstration system and system model will ensure that the system is protected against abuse and that user data is kept private.

6.2.1 Test System

The demonstration system described in Chapter 5 was only a partial implementation of the system described in the preceding two chapters, as detailed in the system scope in Section 5.2.2. The major limitation to testing the full system was the unavailability of 16j WiMAX and DSRC test equipment. Once this equipment becomes available, a complete test system can be developed. Future work on the demonstration system will include performance testing and optimization. Development of DSRC/WiMAX test nodes will allow in-vehicle demonstrations and confirmation of the improvement in system performance. For example, experimentation will be used to set the values of timers within the L3NP protocol, as discussed in Section 4.7.3. The demonstration system detailed in Chapter 5 can be used as a framework and extended to this complete the functionality of the test system.

6.2.2 Computer Modeling

Further system performance measurement can be done through computer modeling of the proposed system. Two tools that were examined for possible system simulation included Network Simulator 2 (ns-2) and OPNET modeller. Both of these software packages are designed to simulate the entire network stack and provide performance measurements, as well as testing a variety of system configurations. While having preliminary support for Mobile WiMAX and some support for DSRC, at the present time nether ns-2 or OPNET have support for the draft IEEE 802.16j amendment, making them unsuitable for modeling. This will likely change once the amendment is approved and released.

Once 16j support is available in these software packages, modeling of the proposed system could prove to be an invaluable tool for calculating the performance of the proposed system and optimizing it. A software simulation environment can be far more versatile in the testing than the demonstration test bed could possibly be. By using both, the demonstration system can provide real-world confirmation of the results found in modeling.

6.2.3 Security Analysis

The proposed WiMAX/DSRC system operates over an inherently insecure wireless broadcast medium and is providing valuable network access services. It is in the interests of both the service provider and the end users for the system to be as secure as possible.

WiMAX performs node authentications during the initial network entry process. It also provides support for security features including frame signing and encryption. The effectiveness of these features will need to be reviewed, as well as any problems implementing them across the distributed access model suggested in both the 16j amendment and the combined WiMAX/DSRC system.

The L3NP also uses authentication during initial network entry, following the model of PPP and derived protocols such as L2TP. A future security feature that should be investigated in L3NP is frame encryption Since L3NP packets are transmitted over a link layer, the IPSec method used with L2TP is not applicable. Instead, an alternative encryption scheme needs to be developed. This would entail a key exchange process during the authentication step and a method for encoding and decoding of the bodies of L3NP frames, as well as detection of frame manipulation.

Appendix A: MATLAB Source Code

This appendix contains the MATLAB programs used for system capacity calculations in Chapter 2.

A.1 WiMAX Capacity Calculations

A.1.1 WimaxCapacity.m

```
% --- The number of connections ---
nc = 60;
% ======================================

% --- Ratio of DL:UL symbols ---
% The total number of symbols per frame should be 47
Symbols_down = 35;
Symbols_up = 12;
% ======================================

% --- Variables that change based on coding scheme ---
% Current values for 5/6 64-QAM
% Code rate
Rc = 3/4    ;
% 2 for BPSK, 4 for 16-QAM, 6 for 64-QAM
Log2M = 6;
% BpS1 should remain constant
BpS1 = 48;
% BpSm depends on the modulation scheme used
BpSm = 240;
% ======================================

% --- These depend on the channel size ---
% Current values for 10 MHz
Nsd_down = 720;
Nsd_up = 560;
Subchan_down = 30;
Subchan_up = 35;
% ======================================

% --- These should remain constant ---
% 102.9 us
Tsymbol = 0.0001029;
% 5 ms
Tframe = 0.005;
% ======================================

% -=- -=- -=- -=- -=- -=- -=- -=- -=- -=-
% Everything beyond this point is math
% -=- -=- -=- -=- -=- -=- -=- -=- -=- -=-

% 2 = PSK, 4 = 16QAM, 6 = 64QAM
Log2M_in = [2 4 6];
BpSm_in = [72 144 216];
```

```
for a = 1:3
    Log2M = Log2M_in(a);
    BpSm = BpSm_in(a);

for n = 2:71
    b = a + 1;
    nc = n - 1;
    Phi_PhyPerSym = (((Symbols_down * Nsd_down * Rc * Log2M) + (Symbols_up *
Nsd_up * Rc * Log2M))/((Symbols_up + Symbols_down) * Tsymbol));
    Phi_PhyPerSym_down = ((Nsd_down * Rc * Log2M) / Tsymbol);
    Phi_PhyPerSym_up = ((Nsd_up * Rc * Log2M) / Tsymbol);


    NSlots_down = (((Symbols_down - 1) * Subchan_down)/2);
    NSlots_up = ((Symbols_up * Subchan_up)/3);

    NPayload_down = (NSlots_down - (((388 + (108 * nc))/BpS1) + (0.5 * nc)));
    NPayload_up = (NSlots_up - (6 + (0.5 * nc)));

    Phi_NetPerMac = (((NPayload_down + NPayload_up) * BpSm) / Tframe);

    Eff = Phi_NetPerMac / Phi_PhyPerSym;

% -=- recorded output -=-
    EffOut(1,n) = nc;
    EffOut(b,1) = Log2M;
    EffOut(b,n) = Eff;
end

end
```

A.2 Clustering Benefit Calculations

A.2.1 SystemBenefit.m

```
% SystemBenefit.m
% Used to demonstrate the decrease in overhead with
% various cluster sizes, as well as determining
% the avaliable system capacity (per node) on both
% the WiMAX and DSRC end of the system

% By Nick C. Doyle
% Last modified 02 Feb 2009

% --- The number of connections ---
nc = 70;
% Effective number of connections
nc_eff = nc;

% ======================================

% --- Ratio of DL:UL symbols ---
% The total number of symbols per frame should be 47
Symbols_down = 35;
Symbols_up = 12;
% ======================================

% --- Variables that change based on coding scheme ---
% Current values for 5/6 64-QAM
% Code rate
Rc = 3/4;
% 2 for BPSK, 4 for 16-QAM, 6 for 64-QAM
Log2M = 6;
% BpS1 should remain constant
BpS1 = 48;
% BpSm depends on the modulation scheme used
BpSm = 216;
% ======================================

% --- These depend on the channel size ---
% Current values for 10 MHz
Nsd_down = 720;
Nsd_up = 560;
Subchan_down = 30;
Subchan_up = 35;
% ======================================

% --- These should remain constant ---
% 102.9 us
Tsymbol = 0.0001029;
% 5 ms
Tframe = 0.005;
% ======================================

% --- Calculations for DSRC capacity ---
Dcapacity_Mbit = 27;
```

```
% The number of nodes in a cluster
Ncluster = 1;

% ======================================

% -=- -=- -=- -=- -=- -=- -=- -=- -=- -=-
% Everything beyond this point is math
% -=- -=- -=- -=- -=- -=- -=- -=- -=- -=-

% 2 = PSK, 4 = 16QAM, 6 = 64QAM
%Log2M_in = [2 4 6];
%BpSm_in = [72 144 216];

% Calculations
% Capacity in bits per second
Dcapacity = Dcapacity_Mbit; %* 1048576;

% Capacity of an individual node
Ncapacity = Dcapacity / Ncluster;

% Adjust the number of active users
for x = 1:14
    %(5,10,15...70)
    nc = (x * 5);

% Vary the cluster size and record the effect on capacity
for n = 1:20
    Ncluster = n;
    % Using a simple calculation to determine a ball-park capacity for DSRC
    Ncapacity = Dcapacity / Ncluster;

    % Calculate the effective nc
    nc_eff = (nc / Ncluster);


    Phi_PhyPerSym = (((Symbols_down * Nsd_down * Rc * Log2M) + (Symbols_up *
Nsd_up * Rc * Log2M))/((Symbols_up + Symbols_down) * Tsymbol));
    Phi_PhyPerSym_down = ((Nsd_down * Rc * Log2M) / Tsymbol);
    Phi_PhyPerSym_up = ((Nsd_up * Rc * Log2M) / Tsymbol);


    NSlots_down = (((Symbols_down - 1) * Subchan_down)/2);
    NSlots_up = ((Symbols_up * Subchan_up)/3);

    NPayload_down = (NSlots_down - (((388 + (108 * nc_eff))/BpS1) + (0.5 *
nc_eff)));
    NPayload_up = (NSlots_up - (6 + (0.5 * nc_eff)));

    Phi_NetPerMac = (((NPayload_down + NPayload_up) * BpSm) / Tframe);

    Eff = Phi_NetPerMac / Phi_PhyPerSym;

    % Capture the "original" efficency to gauge improvement
    if(n == 1)
        Eff_orig = Eff;
    end

    % Calculate Improvement in %
    Eff_imp = (Eff - Eff_orig) * 100;

    % Calculate bit rate per user in the system
    Phi_MacPerUser = (Phi_NetPerMac / nc) / 1048576;

% -=- recorded output -=-
%
    % Calculate index
    z = ((x - 1) * 4) + 3;

    EffOut(1,n) = Ncluster;
    EffOut(2,n) = Ncapacity;
    EffOut(z,n) = nc_eff;
    EffOut(z+1,n) = Eff;
    EffOut(z+2,n) = Eff_imp;
    EffOut(z+3,n) = Phi_MacPerUser;

end

end
```

Appendix B: Demonstration System

This appendix contains the code of the demonstration system described in Chapter 5 of this thesis.

B.1 Shared Library

B.1.1 LinuxTun.c

```
/* LinuxTun.c
 * Description: The interface glue between the C# WiMAXDSRC program and the
 * Linux Tun/Tap kernel module
 *
 * Based on work in the Ipop project at UFlorida
 */

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

#include <sys/ioctl.h>

#include <net/if.h>
#include <linux/if_tun.h>

/* Open the tunnel device. This code is based heavily on the
 * Tun/Tap module documentation */
int TunOpen (char *argRetDev)
{
  struct ifreq ifr;
  int tunFd, err;

  if((tunFd = open("/dev/net/tun", O_RDWR)) < 0)
  {
    perror("Failed to open /dev/net/tun");
    return(-1);
  }

  memset(&ifr, 0, sizeof(ifr));

  // Using a TUN interface
  ifr.ifr_flags = IFF_TUN | IFF_NO_PI;

  // Open the new device
  if(ioctl(tunFd, TUNSETIFF, (void *) &ifr) < 0)
  {
    perror("TUNSETIFF failed");
    close(tunFd);
    return(-1);
  }
  // Return the device name
  strcpy(argRetDev, ifr.ifr_name);
  return(tunFd);
}

/* Close the network device */
int TunClose (int argFd)
{
  return close(argFd);
}

/* Read a message from the Tun device */
int TunRead (int argFd, void *argPacket, int argLength)
{
  return read(argFd, argPacket, argLength);
}

/* Write a message to the Tun device */
int TunWrite (int argFd, u_char *argPacket, int argLength)
{
  return write(argFd, argPacket, argLength);
}

/* Set up IP address */
int TunSetIP (char *argDev, char *argIP)
{
  return 0;
}
```

B.2 Base Program Class

B.2.1 Program.cs

```
/* Name: Program.cs
 * The main entry point into the demonstration system. Allows the user
 * to select which node (SS/RS/BS/IAG) the user wishes to operate as,
 * as well as setting some system parameters. Also includes some generic
 * definitions. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public class Program
    {
        public const int programVersion = 1;

        static void Main(string[] args)
        {
            bool modeSet = false;
            string tempRead;
            int tempMode;
            ModBase mainMod = null;
            string argRSIP;
            const string defArgRSIP = "127.0.0.1";
            string argRSPort;
            const string defArgRSPort = "12127";
            string argBSIP;
            const string defArgBSIP = "127.0.0.1";
            string argBSPort;
            const string defArgBSPort = "12123";
            string argBSID;
            const string defArgBSID = "2000";
            string argIAGIP;
            const string defArgIAGIP = "127.0.0.1";
            string argIAGPort;
            const string defArgIAGPort = "12125";

            Console.WriteLine("-> WiMAX/DSRC - Version {0}", programVersion);
            Console.WriteLine("-> (c)2009 by Nicholas C. Doyle\n");

            while (!modeSet)
            {
                Console.WriteLine("[Program.Main] Select Operation Mode");
                Console.WriteLine("1: IAG, 2: BS, 3: RS, 4: SS (Windows), 5: SS
(Linux) (using defaults)");
                Console.Write("Or '0' to quit\n> ");
                tempRead = Console.ReadLine();
                tempMode = int.Parse(tempRead);

                switch (tempMode)
                {
                    case 0:
                        Console.WriteLine("[Program.Main] Quitting");
                        return;

                    case 1:
                        modeSet = true;
                        Console.WriteLine("[Program.Main] Selected IAG mode
(Defaults)\nIAG port: {0}",
                            defArgIAGPort);

                        argIAGPort = defArgIAGPort;

                        mainMod = new ModIAG(argIAGPort);
                        break;

                    case 2:
                        modeSet = true;
                        Console.WriteLine("[Program.Main] Selected BS mode
(Defaults)\nIAG: {0}:{1}\nBS port: {2}, bsid: {3}",
                            defArgIAGIP, defArgIAGPort, defArgBSPort,
defArgBSID);

                        argIAGIP = defArgIAGIP;
                        argIAGPort = defArgIAGPort;
                        argBSPort = defArgBSPort;
                        argBSID = defArgBSID;

                        mainMod = new ModBS(argIAGIP, argIAGPort, argBSPort,
argBSID);
                        break;

                    case 3:
                        modeSet = true;
                        Console.WriteLine("[Program.Main] Selected RS mode
(Defaults)\nBS: {0}:{1}\nRS port: {2}",
                            defArgBSIP, defArgBSPort, defArgRSPort);

                        argBSIP = defArgBSIP;
                        argBSPort = defArgBSPort;
                        argRSPort = defArgRSPort;

                        mainMod = new ModRS(argBSIP, argBSPort, argRSPort);
                        break;

                    case 4:
                        modeSet = true;
                        Console.WriteLine("[Program.Main] Selected SS mode
(Default Windows Mode)\nRS: {0}:{1}",
                            defArgRSIP, defArgRSPort);
                        argRSIP = defArgRSIP;
                        argRSPort = defArgRSPort;

                        mainMod = new ModSS(argRSIP, argRSPort,
(int)OSType.Windows);
                        break;

                    case 5:
                        modeSet = true;
                        Console.WriteLine("[Program.Main] Selected SS mode
(Default Linux Mode)\nRS: {0}:{1}",
                            defArgRSIP, defArgRSPort);
                        argRSIP = defArgRSIP;
                        argRSPort = defArgRSPort;

                        mainMod = new ModSS(argRSIP, argRSPort,
(int)OSType.Linux);
                        break;

                    default:
                        Console.WriteLine("[Program.Main] Invalid selection.
Please try again.\n");
                        break;
                }
            }

            // Start up the selected module
            mainMod.Run();
        }
    }
    // Generic definitions

    public enum OSType : int
    {
        Unknown,
        Windows,
        Linux
    }

    public enum ReturnVal : int
    {
        OK = 0,
        Error = -1,
        Fail = -2
    }
}
```

B.3 System Modules

B.3.1 ModBase.cs

```
/* Name: ModBase.cs
 * The base class for other module classes in the program. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public abstract class ModBase
    {
        public abstract int Run();

        // Sleep Time for loops. Set at 500ms
        protected static int SleepTime = 500;
    }
}
```

B.3.2 ModSS.cs

```
/* Name: ModSS.cs
 * Impliments the Subscriber Station (SS) node functionality.
 * Launches ModAC and ModTun as threads/subprocesses */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace WiMAXDSRC
{
    public class ModSS : ModBase
    {
        public EntrySS mySS;
        public ModAC myAC;

        public ModSS(string argRSIp, string argRSPort, int argOSMode)
        {
            mySS = new EntrySS();

            mySS.RSIp = argRSIp;
            mySS.RSPort = int.Parse(argRSPort);

            mySS.osMode = argOSMode;

            myAC = new ModAC(ref mySS);
        }

        public override int Run()
        {
            Console.WriteLine("[ModSS.Main] Started ModSS.Run()");

            // 1. Spawn thread to connect with the RS node
            Thread rsCon = new Thread(new ThreadStart(RSConnect));
            rsCon.Start();

            // Start L3np module
            myAC.Run();

            // 2. Loop waiting for commands
            while (true)
            {
                Thread.Sleep(SleepTime);
            }
            // return (int)ReturnVal.OK;
        }

        // Connect to the RS Server
        public void RSConnect()
        {
            Console.WriteLine("[ModSS.RSConnect] Thread Started");

            // Connect to RS
            mySS.wimax = new Wimax(Wimax.netMode.Client, NetBase.netType.SS,
mySS);
            mySS.wimax.Connect(mySS.RSIp, mySS.RSPort);
            mySS.status = (int)Status.Connected;

            // Start up the read handler
            mySS.wimax.AsyncReadStart(mySS, new
AsyncCallback(this.RSReadComplete));

            // The write queue loop
            while (true)
            {
                lock (mySS.sendQueue)
                {

                    if (mySS.sendQueue.Count != 0)
                    {
                        mySS.wimax.Send(mySS.sendQueue.Dequeue());
                    }

                    else if (mySS.AppQueueCount != 0)
                    {
                        byte[] appPayload;
                        Wimax.Frame appFrame;
                        Console.WriteLine("[ModSS.RSConnect] App packet to
send");
                        appPayload = mySS.AppQueue;
                        appFrame = new Wimax.Frame(mySS.transportCid,
appPayload);
                        mySS.wimax.Send(appFrame);
                    }
                }

                Thread.Sleep(SleepTime);
            }
        }

        // Handler for when something comes in to read on the network interface
        private void RSReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntrySS mySS = (EntrySS)ar.AsyncState;
            Wimax.Data myData = new NetBase.Data();
            Wimax.Frame myFrame;

            Console.WriteLine("[ModSS.RSReadComplete] Read something");

            mySS.wimax.AsyncRead(myData, ar, out myFrame);
            Console.WriteLine("[ModSS.RSReadComplete] cid:{0}, length:{1}",
myFrame.cid, myFrame.length);

            RSManage(myFrame, mySS);

            // Start up another receive
            mySS.wimax.AsyncReadStart(mySS, new
AsyncCallback(this.RSReadComplete));
        }

        /* Determine how to handle the received message
         * If the message is a management type, handle it internally.
         * General purpose messages are sent to the Access Concentrator
         * module for processing. */
        public int RSManage(Wimax.Frame argFrame, EntrySS argEntry)
        {

            // Handle management messages
            if (argFrame is Wimax.MgmtFrame)
            {
                Wimax.MgmtFrame myFrame = (Wimax.MgmtFrame)argFrame;
                Wimax.MgmtFrame sendFrame;
                Wimax.Frame demoFrame;
                Console.WriteLine("[ModSS.RSManage] Received {0} bytes, message
{1} from CID {2}",
                    myFrame.length, myFrame.mgmtMsg, myFrame.cid);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.SYNC:
                        argEntry.wimax.SYNCClient(myFrame, out sendFrame);
                        argEntry.wimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.RNG_RSP:
                        argEntry.wimax.RNG_RSPClient(myFrame, out sendFrame);
                        argEntry.wimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.REG_RSP:
                        argEntry.wimax.REG_RSPClient(myFrame, out sendFrame);
                        argEntry.wimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.DSA_RSP:
                        argEntry.wimax.DSA_RSPClient(myFrame, out sendFrame);
                        argEntry.wimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.DSX_RVD:
                        argEntry.wimax.DSX_RVDClient(myFrame);
                        break;

                    default:
                        break;
                }
            }
            else
            {
                // Put the message on the AC queue
                myAC.NetQueue = argFrame.payload;
                //argEntry.wimax.DemoClient(argFrame);
            }

            return (int)ReturnVal.OK;
        }
    }
}
```

B.3.3 ModRS.cs

```
/* Name: ModRS.cs
 * Impliments the Relay Station (RS) node functionality.*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public class ModRS : ModBase
    {
        public EntryRS myRS;

        public ModRS(string argBSIp, string argBSPort, string argRSPort)
        {
            myRS = new EntryRS();

            myRS.BSIp = argBSIp;
            myRS.BSPort = int.Parse(argBSPort);
            myRS.RSPort = int.Parse(argRSPort);
        }

        public override int Run()
        {
            Console.WriteLine("[ModRS.Run] Started ModRS.Run()");

            // Start Thread to connect to BS
            Thread bsCon = new Thread(new ThreadStart(BSConnect));
            bsCon.Start();

            // Start Thread to handle SS connections
            Thread ssCon = new Thread(new ThreadStart(SSConnect));
            ssCon.Start();


            // Loop for commands
            while (true)
            {
                Thread.Sleep(SleepTime);
            }

            //return (int)ReturnVal.OK;
        }

        // Connect to an SS clients
        public void SSConnect()
        {
            myRS.listenWimax = new Wimax(NetBase.netMode.Server,
NetBase.netType.RS, myRS);
            myRS.listenWimax.Listen(myRS.RSPort, this.SSHandler);
        }

        // Handler for individual connections
        public void SSHandler(object argData)
        {
            Console.WriteLine("[ModRS.SSConnect] Starting new handler");

            // Don't process anything until connected to the BS
            while (myRS.status != (int)Status.Running)
            {
                Thread.Sleep(SleepTime);
            }

            EntrySS mySS = new EntrySS();
            mySS.socket = (Socket)argData;
            LinkedListNode<EntrySS> myNode;

            // Add to the list of clients
            lock (myRS.ssConnections)
            {
                myNode = myRS.ssConnections.AddLast(mySS);
                Console.WriteLine("[ModRS.SSConnect] Added entry, total: {0}",
myRS.ssConnections.Count);
            }

            mySS.wimax = new Wimax(NetBase.netMode.Relay, NetBase.netType.SS,
mySS, mySS.socket);
            mySS.status = (int)Status.Connected;

            // Start up read handler
            mySS.wimax.AsyncReadStart(mySS, new
AsyncCallback(this.SSReadComplete));

            // Send initial connect message
            Wimax.MgmtFrame startFrame;
            mySS.wimax.SYNCRelay(out startFrame, myRS);
            mySS.wimax.AsyncSend(startFrame);

            // The write queue loop
            while (true)
            {
                lock (mySS.sendQueue)
                {
                    if (mySS.sendQueue.Count != 0)
                    {
                        mySS.wimax.Send(mySS.sendQueue.Dequeue());
                    }
                }
                Thread.Sleep(SleepTime);
            }
        }

        // Read handler for incoming messages from the SS node
        private void SSReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntrySS mySS = (EntrySS)ar.AsyncState;
            Wimax.Data myData = new NetBase.Data();
            Wimax.Frame myFrame;

            Console.WriteLine("[ModRS.SSReadComplete] Received something");

            mySS.wimax.AsyncRead(myData, ar, out myFrame);
            SSManage(myFrame, mySS);

            // Start up another receive
            mySS.wimax.AsyncReadStart(mySS, new
AsyncCallback(this.SSReadComplete));
        }

        // Manage incoming messages from the SS node
        public int SSManage(Wimax.Frame argFrame, EntrySS argEntry)
        {

            // Handle management messages
            if (argFrame is Wimax.MgmtFrame)
            {
                Wimax.MgmtFrame myFrame = (Wimax.MgmtFrame)argFrame;
                //Wimax.MgmtFrame sendFrame;

                Console.WriteLine("[ModRS.SSManage] Received {0} bytes, message
{1} from CID {2}",
                    myFrame.length, myFrame.mgmtMsg, myFrame.cid);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.RNG_REQ:
                        argEntry.wimax.RNG_REQRelay(myFrame, myRS);

                        // relay the message
                        myRS.bsWimax.RelayQueue(myFrame, myRS);
                        break;

                    case (int)Wimax.mgmtType.REG_REQ:
                        argEntry.wimax.REG_REQRelay(myFrame, myRS);

                        // Relay the message
                        myRS.bsWimax.RelayQueue(myFrame, myRS);
                        break;

                    case (int)Wimax.mgmtType.DSA_REQ:
                        argEntry.wimax.DSA_REQRelay(myFrame, myRS);
                        // Relay the message
                        myRS.bsWimax.RelayQueue(myFrame, myRS);
                        break;

                    case (int)Wimax.mgmtType.DSA_ACK:
                        argEntry.wimax.DSA_ACKRelay(myFrame, myRS);
                        // Relay the message
                        myRS.bsWimax.RelayQueue(myFrame, myRS);
                        break;

                    default:
                        break;
                }
            }
            // Handle data transmissions
            else
            {
                Console.WriteLine("[ModRS.SSManage] Relaying data message");
                myRS.bsWimax.RelayQueue(argFrame, myRS);
            }

            return (int)ReturnVal.OK;

        }

        // Connect to the BS server
        public void BSConnect()
        {

            Console.WriteLine("[ModRS.BSConnect] Starting Thread");

            // Connect to BS
            myRS.bsWimax = new Wimax(Wimax.netMode.Client, NetBase.netType.RS, 
myRS);
            myRS.bsWimax.Connect(myRS.BSIp, myRS.BSPort);
            myRS.status = (int)Status.Connected;

            // Start up the read handler
            myRS.bsWimax.AsyncReadStart(myRS, new 
AsyncCallback(this.BSReadComplete));

            // The write queue loop
            while (true)
            {
                lock (myRS.sendQueue)
                {
                    // Priority for this node's own traffic
                    if (myRS.sendQueue.Count != 0)
                    {
                        myRS.bsWimax.Send(myRS.sendQueue.Dequeue());
                    }
                    // Check to see if we have stuff to relay
                    else if (myRS.relayQueue.Count != 0)
                    {
                        myRS.bsWimax.RelaySend(myRS);
                    }
                }

                Thread.Sleep(SleepTime);
            }
        }

        // Handler for messages from the RS node
        private void BSReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntryRS myRS = (EntryRS)ar.AsyncState;
            Wimax.Data myData = new NetBase.Data();
            Wimax.Frame myFrame;
            Console.WriteLine("[ModRS.BSReadComplete] Read something");

            myRS.bsWimax.AsyncRead(myData, ar, out myFrame);

            BSManage(myFrame, myRS);

            // Start up another receive
            myRS.bsWimax.AsyncReadStart(myRS, new
AsyncCallback(this.BSReadComplete));
        }

        // Manage traffic from the BS node
        public int BSManage(Wimax.Frame argFrame, EntryRS argEntry)
        {

            // Handle management messages
            if (argFrame is Wimax.MgmtFrame)
            {
                Wimax.MgmtFrame myFrame = (Wimax.MgmtFrame)argFrame;
                Wimax.MgmtFrame sendFrame;

                Console.WriteLine("[ModRS.BSManage] Received {0} bytes, message
{1} from CID {2}",
                    myFrame.length, myFrame.mgmtMsg, myFrame.cid);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.SYNC:

                        argEntry.bsWimax.SYNCClient(myFrame, out sendFrame);
                        argEntry.bsWimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.RNG_RSP:

                        argEntry.bsWimax.RNG_RSPClient(myFrame, out sendFrame);
                        argEntry.bsWimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.REG_RSP:
                        argEntry.bsWimax.REG_RSPClient(myFrame);
                        break;

                    default:
                        break;
                }
            }
            // Process tunneled messages (for SS nodes).
            else if (argFrame is Wimax.TunFrame)
            {
                // Handled a tunneled message
                Wimax.Frame[] frameList;

                // Decode messages
                myRS.bsWimax.RelayRead((Wimax.TunFrame)argFrame, out frameList);
                Console.WriteLine("[ModRS.BSManage] Received relay message with
{0} entries from CID {1}",
                    frameList.Length, argFrame.cid);

                foreach (Wimax.Frame f in frameList)
                {
                    EntrySS mySS;
                    Console.WriteLine("[ModRS.BSManage] Relay message for CID
{0}, length {1}", f.cid, f.length);

                    // Special case for the default CID
                    if (f.cid == Wimax.defaultCid)
                    {
                        argEntry.listenWimax.RelayFindRanging(f,
argEntry.ssConnections, out mySS);

                        // Process
                    }
                    else
                    {
                        argEntry.listenWimax.RelayFindSS(f.cid,
argEntry.ssConnections, out mySS);
                    }

                    if (mySS != null)
                    {
                        BSRelayManage(f, argEntry, mySS);
                    }
                }

            }
            else
            {

                // Handle data transmissions
            }

            return (int)ReturnVal.OK;
        }

        // Manage messages being relayed to an SS node
        public int BSRelayManage(Wimax.Frame argFrame, EntryRS myRS, EntrySS
mySS)
        {

            // Handle management messages
            if ((argFrame.cid == Wimax.defaultCid) || (argFrame.cid ==
mySS.basicCid) ||
                (argFrame.cid == mySS.primaryCid))
            {
                Wimax.MgmtFrame myFrame = new Wimax.MgmtFrame();
                myFrame.ConvertFrame(argFrame);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.RNG_RSP:

                        mySS.wimax.RNG_RSPRelay(myFrame, myRS);
                        mySS.wimax.AsyncSend(myFrame);
                        break;

                    case (int)Wimax.mgmtType.REG_RSP:

                        mySS.wimax.REG_RSPRelay(myFrame, myRS);
                        mySS.wimax.AsyncSend(myFrame);
                        break;

                    case (int)Wimax.mgmtType.DSA_RSP:
                        mySS.wimax.DSA_RSPRelay(myFrame, myRS);
                        mySS.wimax.AsyncSend(myFrame);
                        break;

                    case (int)Wimax.mgmtType.DSX_RVD:
                        mySS.wimax.DSX_RVDRelay(myFrame, myRS);
                        mySS.wimax.AsyncSend(myFrame);
                        break;


                    default:
                        break;
                }
            }
            // Handle data messages
            else
            {
                // Relay message
                mySS.wimax.AsyncSend(argFrame);
            }

            return (int)ReturnVal.OK;
        }
    }
}
```

B.3.4 ModBS.cs

```
/* Name: ModBS.cs
 * Impliments the Base Station (BS) node functionality.*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    class ModBS : ModBase
    {
        public EntryBS myBS = new EntryBS();

        public ModBS(string argIAGIP, string argIAGPort, string argBSPort,
string argBSID)
        {
            myBS.iagIp = argIAGIP;
            myBS.iagPort = int.Parse(argIAGPort);
            myBS.bsPort = int.Parse(argBSPort);
            myBS.bsid = int.Parse(argBSID);

            myBS.myRes = new WimaxRes(myBS.bsid);
        }

        public override int Run()
        {
            Console.WriteLine("[ModBS.Main] Started ModBS.Run()");

            // Create SS entry


            // Connect to the IAG server
            Thread iagCon = new Thread(new ThreadStart(IAGConnect));
            iagCon.Start();

            // Start thread to handle RS connections
            Thread rsCon = new Thread(new ThreadStart(RSConnect));
            rsCon.Start();

            // 2. Loop waiting for commands
            while (true)
            {
                Thread.Sleep(SleepTime);
            }
            // return (int)ReturnVal.OK;
        }

        // Connect to the IAG Server
        public void IAGConnect()
        {
            Console.WriteLine("[ModBS.IAGConnect] Thread Started");

            // Connect to IAG
            myBS.iagNetback = new NetBack(NetBase.netMode.Client,
NetBase.netType.BS, myBS);
            myBS.iagNetback.Connect(myBS.iagIp, myBS.iagPort);

            // Start up the read handler
            myBS.iagNetback.AsyncReadStart(myBS, new
AsyncCallback(this.IAGReadComplete));

            // Send a handshake message
            NetBack.Frame sendFrame;
            myBS.iagNetback.HS_REQClient(out sendFrame);
            myBS.iagNetback.AsyncSend(sendFrame, myBS);
            // Do the write queue loop.
            while (true)
            {
                lock (myBS.sendQueue)
                {
                    if (myBS.sendQueue.Count != 0)
                    {
                        myBS.iagNetback.Send(myBS.sendQueue.Dequeue());
                    }
                }
                Thread.Sleep(SleepTime);
            }
        }

        // Handler for messages received from the IAG
        private void IAGReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntryBS myBS = (EntryBS)ar.AsyncState;
            NetBack.Data myData = new NetBase.Data();
            NetBack.Frame myFrame = new NetBack.Frame();

            myBS.iagNetback.AsyncRead(myData, myBS, ar, myFrame);

            Console.WriteLine("[ModBS.IAGReadComplete] Received {0} bytes, type 
{1} from CID {2}",
                myFrame.length, myFrame.type, myFrame.cid);

            IAGManage(myFrame, myBS);

            // Start up another receive
            myBS.iagNetback.AsyncReadStart(myBS, new 
AsyncCallback(this.IAGReadComplete));
        }

        // Manage message from the IAG node
        public int IAGManage(NetBack.Frame argFrame, EntryBS argEntry)
        {
            switch (argFrame.type)
            {
                // Read handshake reply
                case (int)NetBack.MessageType.HS_RSP:
                    argEntry.iagNetback.HS_RSPClient(argFrame);
                    break;

                case (int)NetBack.MessageType.DATA:
                    IAGDataManage(argFrame, argEntry);

                    break;


                default:
                    break;
            }
            return (int)ReturnVal.OK;
        }

        //  Data received for an SS node. Find them and relay the message.
        protected int IAGDataManage(NetBack.Frame argFrame, EntryBS argEntry)
        {
            // Find user
            EntrySS mySS;

            argEntry.listenWimax.RelayFindSS(argFrame.cid,
argEntry.ssConnections, out mySS);
            // Process the frame

            if (mySS == null)
            {
                //
                return (int)ReturnVal.Fail;

            }

            Console.WriteLine("[ModBS.IAGManage] Forwading message to RS");
            Wimax.Frame sendFrame = new Wimax.Frame(argFrame.cid,
argFrame.payload);
            return mySS.servingRS.bsWimax.RelayQueue(sendFrame, mySS.servingRS);

        }

        // Connect to RS nodes
        public void RSConnect()
        {
            myBS.listenWimax = new Wimax(NetBase.netMode.Server,
NetBase.netType.BS, myBS);
            myBS.listenWimax.Listen(myBS.bsPort, this.RSHandler);
        }

        // Handler for individual connections
        public void RSHandler(object argData)
        {
            Console.WriteLine("[ModBS.RSHandler] Starting new handler");

            EntryRS myRS = new EntryRS();
            myRS.bsSocket = (Socket)argData;
            LinkedListNode<EntryRS> myNode;

            // Add to the list of clients
            lock (myBS.rsConnections)
            {
                myNode = myBS.rsConnections.AddLast(myRS);
                Console.WriteLine("[ModBS.RSHandler] Added entry, total: {0}",
myBS.rsConnections.Count);
            }

            // We don't know if this is an RS or a BS at this point, which is
why type is unknown
```

```
            // We'll use EntryRS for now, since we're not modeling direct SS
connections yet
            // (replace later with a generic handler)
            myRS.bsWimax = new Wimax(NetBase.netMode.Server,
NetBase.netType.Unknown, myRS, myRS.bsSocket);
            myRS.status = (int)Status.Connected;

            // Start up read handler
            myRS.bsWimax.AsyncReadStart(myRS, new
AsyncCallback(this.RSReadComplete));

            // Send initial connect message
            Wimax.MgmtFrame startFrame;
            myRS.bsWimax.SYNCServer(out startFrame, myBS);
            myRS.bsWimax.AsyncSend(startFrame);

            // The write queue loop
            while (true)
            {
                lock (myRS.sendQueue)
                {
                    // Priority for direct connection messages
                    if (myRS.sendQueue.Count != 0)
                    {
                        myRS.bsWimax.Send(myRS.sendQueue.Dequeue());
                    }
                    // Check if we have any messages to tunnel
                    else if (myRS.relayQueue.Count != 0)
                    {
                        myRS.bsWimax.RelaySend(myRS);
                    }
                }
                Thread.Sleep(SleepTime);
            }
        }

        // Read handler for messages from the RS node
        private void RSReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntryRS myRS = (EntryRS)ar.AsyncState;
            Wimax.Data myData = new Wimax.Data();
            Wimax.Frame myFrame;

            Console.WriteLine("[ModBS.RSReadComplete] Received something");
            myRS.bsWimax.AsyncRead(myData, ar, out myFrame);

            RSManage(myFrame, myRS);

            // Start up another receive
            myRS.bsWimax.AsyncReadStart(myRS, new
AsyncCallback(this.RSReadComplete));
        }

        // Process message from an RS node
        public int RSManage(Wimax.Frame argFrame, EntryRS argEntry)
        {
            // Handle management messages
            if (argFrame is Wimax.MgmtFrame)
            {
                Wimax.MgmtFrame myFrame = (Wimax.MgmtFrame)argFrame;
                Wimax.MgmtFrame sendFrame;

                Console.WriteLine("[ModBS.RSManage] Received {0} bytes, message
{1} from CID {2}",
                    myFrame.length, myFrame.mgmtMsg, myFrame.cid);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.RNG_REQ:
                        argEntry.bsWimax.RNG_REQServer(myFrame, out sendFrame,
myBS);
                        argEntry.bsWimax.AsyncSend(sendFrame);
                        break;

                    case (int)Wimax.mgmtType.REG_REQ:
                        argEntry.bsWimax.REG_REQServer(myFrame, out sendFrame,
myBS);
                        argEntry.bsWimax.AsyncSend(sendFrame);
                        break;

                    default:
                        break;
                }
            }
            else if (argFrame is Wimax.TunFrame)
            {
                Wimax.Frame[] frameList;

                // Decode messages
                myBS.listenWimax.RelayRead((Wimax.TunFrame)argFrame, out
frameList);

                Console.WriteLine("[ModBS.RSManage] Received relay message with
{0} entries from CID {1}",
                    frameList.Length, argFrame.cid);

               // All frames are plain frames and need to be decoded
                foreach (Wimax.Frame f in frameList)
                {
                    if (f.cid == Wimax.defaultCid)
                    {
                        Console.WriteLine("[ModBS.RSManage] Processing new
user");
                        // special new connection stuff
                        SSAddUser(f, argEntry);
                    }
                    else
                    {
                        // already connected user stuff
                        SSManage(f, argEntry);
                    }
                }

            }
            else
            {
                // Data messages
            }

            return (int)ReturnVal.OK;
        }

        // Create new table entry for SS user
        protected int SSAddUser(Wimax.Frame argFrame, EntryRS argRS)
        {
            Wimax.MgmtFrame myFrame = new Wimax.MgmtFrame();

            // Convert argFrame to MgmtFrame
            myFrame.ConvertFrame(argFrame);


            // Only interested in RNG_REQ using the default CID

            if ((myFrame.mgmtMsg != (int)Wimax.mgmtType.RNG_REQ))
            {
                Console.WriteLine("[ModBS.SSAddUser] Not a RNG_REQ message");
                return (int)ReturnVal.Fail;
            }

            Console.WriteLine("[ModBS.SSAddUser] Adding new user");

            // create new entry
            EntrySS mySS = new EntrySS();
            mySS.wimax = new Wimax(NetBase.netMode.Server, NetBase.netType.SS,
mySS);
            mySS.servingRS = argRS;

            // Add to database
            lock(myBS.ssConnections)
            {
                myBS.ssConnections.AddLast(mySS);
            }

            // Process the RNG_REQ message
            Wimax.MgmtFrame sendFrame;
            mySS.wimax.RNG_REQServer(myFrame, out sendFrame, myBS);

            argRS.bsWimax.RelayQueue(sendFrame, argRS);

            return (int)ReturnVal.OK;
        }

        // Process management messages for SS user
        protected int SSManage(Wimax.Frame argFrame, EntryRS argRS)
```

```
        {
            // Find user
            EntrySS mySS;

            argRS.bsWimax.RelayFindSS(argFrame.cid, myBS.ssConnections, out
mySS);
            // Process the frame

            if (mySS == null)
            {
                //
                return (int)ReturnVal.Fail;
            }

            // Process management messages
            if ((argFrame.cid == mySS.basicCid) || (argFrame.cid ==
mySS.primaryCid))
            {
                Wimax.MgmtFrame sendFrame;
                Wimax.MgmtFrame myFrame = new Wimax.MgmtFrame();
                myFrame.ConvertFrame(argFrame);

                switch (myFrame.mgmtMsg)
                {
                    case (int)Wimax.mgmtType.REG_REQ:
                        mySS.wimax.REG_REQServer(myFrame, out sendFrame, myBS);

                        argRS.bsWimax.RelayQueue(sendFrame, argRS);
                        break;

                    case (int)Wimax.mgmtType.DSA_REQ:
                        mySS.wimax.DSA_REQServer(myFrame, out sendFrame, myBS);
                        argRS.bsWimax.RelayQueue(sendFrame, argRS);

                        mySS.wimax.DSA_RSPServer(myFrame, out sendFrame, myBS);
                        argRS.bsWimax.RelayQueue(sendFrame, argRS);
                        break;

                    case (int)Wimax.mgmtType.DSA_ACK:
                        mySS.wimax.DSA_ACKServer(myFrame, myBS);
                        break;

                    default:
                        break;
                }
            }
            // Handle data messages
            else
            {
                Console.WriteLine("[ModBS.SSManage] Forwading message to IAG");
                NetBack.Frame sendFrame = new
NetBack.Frame((int)NetBack.MessageType.DATA, argFrame.cid, argFrame.payload);
                myBS.iagNetback.AsyncSend(sendFrame, myBS);
            }
            return (int)ReturnVal.OK;
        }
    }
}
```

B.3.5 ModIAG.cs

```
/* Name: ModIAG.cs
 * Impliments the Internet Access Gateway (IAG) node functionality.
 * Launches ModNS as a thread/subprocess */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    class ModIAG : ModBase
    {
        EntryIAG myIAG;
        ModNS myNS;

        public ModIAG(string argIAGPort)
        {
            myIAG = new EntryIAG();
            myIAG.iagPort = int.Parse(argIAGPort);

            myNS = new ModNS(ref myIAG);
        }

        public override int Run()
        {
            Console.WriteLine("[ModIAG.Run] Started ModIAG.Run()");

            // Start Thread to handle BS connections
            Thread bsCon = new Thread(new ThreadStart(BSConnect));
            bsCon.Start();

            // Start application layer queue listener
            Thread iagListen = new Thread(new ThreadStart(IAGListen));
            iagListen.Start();

            // Start NS node
            myNS.Run();

            // Loop for commands
            while (true)
            {
                Thread.Sleep(SleepTime);
            }

            // return (int)ReturnVal.OK;
        }
        // Listen for BS clients
        public void BSConnect()
        {
            myIAG.netback = new NetBack(NetBack.netMode.Server,
NetBack.netType.IAG, null);
            myIAG.netback.Listen(myIAG.iagPort, this.BSHandler);
        }

        // Listen for data from application layer
        public void IAGListen()
        {
            while (true)
            {
                while (myIAG.AppQueueCount > 0)
                {
                    ModNS.InteropFrame appFrame;
                    EntryBS myBS;
                    appFrame = myIAG.AppQueue;

                    Console.WriteLine("[ModIAG.IAGListen] Received data from app
layer");
                    Console.WriteLine("bsid: {0}, cid: {1}", appFrame.bsid,
appFrame.cid);

                    // Find the relevent BS connection
                    myIAG.FindBS(appFrame.bsid, out myBS);
                    // Put it onto the send queue
                    myBS.iagNetback.AsyncSend(new
NetBack.Frame((int)NetBack.MessageType.DATA,
                        appFrame.cid, appFrame.payload), myBS);
                }

                Thread.Sleep(SleepTime);
            }
        }


        // Handler for individual connections
        public void BSHandler(object argData)
        {
            Console.WriteLine("[ModIAG.BSHandler] Starting new handler");

            // Don't process anything until connected to the BS

            EntryBS myBS = new EntryBS();
            myBS.iagSocket = (Socket)argData;

            // Add to the database
            lock (myIAG.bsConnections)
            {
                LinkedListNode<EntryBS> myNode =
myIAG.bsConnections.AddLast(myBS);
            }
            myBS.iagNetback = new NetBack(NetBack.netMode.Server,
NetBack.netType.BS, myBS, myBS.iagSocket);

            // Start read handler
            myBS.iagNetback.AsyncReadStart(myBS, new
AsyncCallback(this.BSReadComplete));

            //myBS.iagNetback.Handshake(myBS);

            // Do the write queue loop.
            while (true)
            {
                lock (myBS.sendQueue)
                {
                    if (myBS.sendQueue.Count != 0)
                    {
                        myBS.iagNetback.Send(myBS.sendQueue.Dequeue());
                    }
                }

                Thread.Sleep(SleepTime);
            }
        }

        // Read handler for message from BS node
        private void BSReadComplete(IAsyncResult ar)
        {
            // Decode the message
            EntryBS myBS = (EntryBS)ar.AsyncState;
            NetBack.Data myData = new NetBase.Data();
            NetBack.Frame myFrame = new NetBack.Frame();

            myBS.iagNetback.AsyncRead(myData, myBS, ar, myFrame);

            Console.WriteLine("[ModIAG.RSReadComplete] Received {0} bytes, type
{1} from CID {2}",
                myFrame.length, myFrame.type, myFrame.cid);

            BSManage(myFrame, myBS);

            // Start up another receive
            myBS.iagNetback.AsyncReadStart(myBS, new
AsyncCallback(this.BSReadComplete));
        }

        // Manage message received from BS node
        public int BSManage(NetBack.Frame argFrame, EntryBS argEntry)
        {
            NetBack.Frame sendFrame;

            switch (argFrame.type)
            {
                // Received handshake request. Send reply.
                case (int)NetBack.MessageType.HS_REQ:
                    argEntry.iagNetback.HS_REQServer(argFrame, out sendFrame, 
ref myIAG);
                    argEntry.iagNetback.AsyncSend(sendFrame, argEntry);
                    break;

                case (int)NetBack.MessageType.DATA:

                    ModNS.InteropFrame NSFrame = new ModNS.InteropFrame();
                    NSFrame.payload = argFrame.payload;
                    NSFrame.cid = argFrame.cid;
                    NSFrame.bsid = argEntry.bsid;

                    myNS.NetQueue = NSFrame;

                    break;

                default:
                    break;
            }

            return (int)ReturnVal.OK;
        }
    }
}
```

B.4 Network Protocol Modules

B.4.1 ModAC.cs

```
/* Name: ModAC.cs
 * Impliments the Access Concentrator (AC) module functionality.
 * Functions as a subprocess within the SS node. Interfaces with
 * ModTun in order to access user network traffic */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace WiMAXDSRC
{
    public class ModAC : ModBase
    {
        protected EntrySS mySS;
        protected L3np myL3np;
        protected ModTun myTun = null;

        protected Queue<byte[]> netQueue;
        protected Queue<byte[]> tunQueue;

        public ModAC(ref EntrySS argSS)
        {
            mySS = argSS;
            myL3np = new L3np(mySS);
            netQueue = new Queue<byte[]>();
            tunQueue = new Queue<byte[]>();
        }

        public override int Run()
        {
            Thread rsCon = new Thread(new ThreadStart(ACMain));
            rsCon.Start();

            return (int)ReturnVal.OK;
        }

        public void ACMain()
        {
            Console.WriteLine("[ModAC.ACMain] Started AC module");

            // Don't operate until we're connected
            while (mySS.status != (int)Status.Running)
            {
                Thread.Sleep(SleepTime);
            }

            // Open up the Tun module if required
            if (mySS.osMode == (int)OSType.Linux)
            {
                myTun = new ModTun(this);
                myTun.Run();
            }

            // Prepare initial message
            L3np.LCP.LCPFrame startFrame;
            myL3np.Lcp.ConfigureRequestClient(out startFrame);
            // Send to SS for processing
            mySS.AppQueue = startFrame.byteArray;

            // The processing loop

            while (true)
            {
                // Messages from the network layer to process
                while (netQueue.Count > 0)
                {
                    NetProc(NetQueue);
                }

                // Messages from the application layer to process
                while (tunQueue.Count > 0)
                {
                    TunProc(TunQueue);
                }

                Thread.Sleep(SleepTime);
            }
        }

        // Process traffic received from ModTun
```

```
        protected int TunProc(byte[] argData)
        {
            // Prepare a new L3np IP frame
            L3np.IP.IPFrame newIP = new L3np.IP.IPFrame();
            newIP.payload = argData;
            Console.WriteLine("[ModAC.TunProc] IP frame has a length of {0}",
newIP.payload.Length);
            newIP.Encode();

            // Put it on the send queue
            mySS.AppQueue = newIP.byteArray;
            Console.WriteLine("[ModAC.TunProc] Packaged an IP frame");

            return (int)ReturnVal.OK;
        }

        // Process traffic received from ModSS
        protected int NetProc(byte[] argData)
        {
            // Decode the header
            L3np.PPPFrame genFrame = new L3np.PPPFrame();

            genFrame.byteArray = argData;
            genFrame.Decode();

            Console.WriteLine("[ModAC.NetProc] Received new L3np message");
            Console.Write("session id: {0}, protocol: {1} ",
                genFrame.sessionid, genFrame.protocol);

            // Reaction based on protocol
            switch (genFrame.protocol)
            {
                case (int)L3np.Protocols.LCP:
                    Console.WriteLine("(LCP)");
                    LCPProc(argData);
                    break;

                case (int)L3np.Protocols.IPCP:
                    Console.WriteLine("(IPCP)");
                    IPCPProc(argData);
                    break;

                case (int)L3np.Protocols.IP:
                    Console.WriteLine("(IP)");
                    IPProc(argData);
                    break;

                default:
                    Console.WriteLine("(Unknown)");
                    break;
            }
            return (int)ReturnVal.OK;
        }

        // Process IP packets - send to ModTun
        protected int IPProc(byte[] argData)
        {
            L3np.IP.IPFrame ipFrame = new L3np.IP.IPFrame();
            ipFrame.byteArray = argData;
            ipFrame.Decode();

            // If we're operating in Linux mode,
            // forward the packet to the tun module
            if (myTun != null)
            {
                Console.WriteLine("[ModAC.IPProc] Payload length: {0}",
ipFrame.payload.Length);
                myTun.ACQueue = ipFrame.payload;
                Console.WriteLine("[ModAC.IPProc] Forwarded IP message to the
tun module");

            }

            return (int)ReturnVal.OK;
        }

        // Process LCP messages
        protected int LCPProc(byte[] argData)
        {
            L3np.LCP.LCPFrame sendFrame;
            L3np.IPCP.IPCPFrame IPCPFrame;
            L3np.LCP.LCPFrame lcpFrame = new L3np.LCP.LCPFrame();
            lcpFrame.byteArray = argData;
            lcpFrame.Decode();

            switch (lcpFrame.code)
            {
                case (int)L3np.Codes.ConfigureAck:
                    IPCPFrame = new L3np.IPCP.IPCPFrame();
                    // Get acknowledgement
                    myL3np.Lcp.ConfigureAckClient(lcpFrame);

                    // Since we were acknowledged, we can start the IP
negotiation process
                    myL3np.Ipcp.ConfigureRequestClient(out IPCPFrame);
                    mySS.AppQueue = IPCPFrame.byteArray;

                    break;

                default:
                    break;
            }

            return (int)ReturnVal.OK;
        }

        // Process IPCP messages
        protected int IPCPProc(byte[] argData)
        {
            L3np.IPCP.IPCPFrame ipcpFrame = new L3np.IPCP.IPCPFrame();
            ipcpFrame.byteArray = argData;
            ipcpFrame.Decode();

            switch (ipcpFrame.code)
            {
                case (int)L3np.Codes.ConfigureAck:
                    // Get acknowledgement
                    myL3np.Ipcp.ConfigureAckClient(ipcpFrame);

                    // At this point, we're ready for transmissions
                    break;

                default:
                    break;
            }

            return (int)ReturnVal.OK;

        }

        // AppProc not implimented
        protected int AppProc(NetBack.Data argData)
        {
            //
            return (int)ReturnVal.Error;
        }


        // The property to allow ModSS to access the incoming message queue
        public byte[] NetQueue
        {
            get
            {
                byte[] tempData;
                lock (netQueue)
                {
                    tempData = netQueue.Dequeue();
                    Console.WriteLine("[ModAC.NetQueue.get] Removed from queue.
Total: {0}", netQueue.Count);

                }
                return tempData;
            }

            set
            {
                lock (netQueue)
                {
                    netQueue.Enqueue(value);
                    Console.WriteLine("[ModAC.NetQueue.set] Added to queue.
Total: {0}", netQueue.Count);
                }
            }
        }
        // The property to allow ModTun to access the incoming message queue.
        public byte[] TunQueue
        {
            get
            {
                byte[] tempData;
                lock (tunQueue)
                {
                    tempData = tunQueue.Dequeue();
                    Console.WriteLine("[ModAC.TunQueue.get] Removed from queue.
Total: {0}", tunQueue.Count);

                }
                return tempData;
            }

            set
            {
                lock (tunQueue)
                {
                    tunQueue.Enqueue(value);
                    Console.WriteLine("[ModAC.TunQueue.set] Added to queue.
Total: {0}", tunQueue.Count);
                }
            }
        }
    }
}
```

B.4.2 ModNS.cs

```
/* Name: ModNS.cs
 * Impliments the Network Server (NS) module functionality.
 * Functions as a subprocess within the IAG node. Will route
 * traffic to/from the internet, but currently echos user
 * traffic back to source. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace WiMAXDSRC
{
    public class ModNS : ModBase
    {
        protected EntryIAG myIAG;
        protected L3np myL3np;

        protected Queue<InteropFrame> netQueue = new Queue<InteropFrame>();
        protected Queue<InteropFrame> appQueue = new Queue<InteropFrame>();

        protected LinkedList<ACEntry> userList = new LinkedList<ACEntry>();

        public ModNS(ref EntryIAG argIAG)
        {
            myIAG = argIAG;
            myL3np = new L3np(myIAG);
            netQueue = new Queue<InteropFrame>();
        }

        public override int Run()
        {
            Thread rsCon = new Thread(new ThreadStart(NSMain));
            rsCon.Start();

            return (int)ReturnVal.OK;
        }

        public void NSMain()
        {
            Console.WriteLine("[ModNS.NSMain] Started NS module");

            // The processing loop
            while (true)
            {
                // Messages from the network layer to process
                while (netQueue.Count > 0)
                {
                    NetProc(NetQueue);
                }

                // Messages from the application layer to process
                /*while (appQueue.Count > 0)
                {
                    AppProc(myL3np.NetQueue);
                }*/

                Thread.Sleep(SleepTime);
            }
        }

        // Process traffic received from the IAG node
        protected int NetProc(InteropFrame argFrame)
        {
            // Decode the header
            L3np.PPPFrame genFrame = new L3np.PPPFrame();
            ACEntry myEntry;

            genFrame.byteArray = argFrame.payload;
            genFrame.Decode();

            Console.WriteLine("[ModNS.NetProc] Frame Received");
            Console.WriteLine("bsid: {0}, cid: {1}, session id: {2}, protocol:
{3}",
                argFrame.bsid, argFrame.cid, genFrame.sessionid,
genFrame.protocol);

            // Send message back
            //myIAG.AppQueue = argFrame;
            // Handle initial connections
            if (genFrame.sessionid == L3np.hsSid)
            {
                NewAC(argFrame.bsid, argFrame.cid, out myEntry);
            }
            else
            {
                FindAC(genFrame.sessionid, out myEntry);
            }

            // Reaction based on protocol
            switch (genFrame.protocol)
            {
                case (int)L3np.Protocols.LCP:
                    Console.WriteLine("(LCP)");
                    LCPProc(argFrame, ref myEntry);
                    break;

                case (int)L3np.Protocols.IPCP:
                    Console.WriteLine("(IPCP)");
                    IPCPProc(argFrame, ref myEntry);
                    break;

                case (int)L3np.Protocols.IP:
                    Console.WriteLine("(IP)");
                    IPProc(argFrame, ref myEntry);
                    break;

                default:
                    Console.WriteLine("(Unknown)");
                    break;
            }
            return (int)ReturnVal.OK;
        }

        /* Process IP traffic. For the time being, we send the message
         * back to the source node in a new L3NP frame */
        protected int IPProc(InteropFrame argFrame, ref ACEntry argAC)
        {
            L3np.IP.IPFrame ipFrame = new L3np.IP.IPFrame();
            InteropFrame retFrame;

            ipFrame.byteArray = argFrame.payload;
            ipFrame.Decode();

            if (ipFrame.payload == null)
            {
                Console.WriteLine("[ModNS.IPProc] ipFrame.payload is null - 
something is broken.");
            }
            else
            {
                Console.WriteLine("[ModNS.IPProc] ipFrame.payload is {0} bytes", 
ipFrame.payload.Length);
```

```
            }

            // At this point, we don't do anything except send the frame back
            retFrame = new InteropFrame(argAC.cid, argAC.bsid,
ipFrame.byteArray);
            myIAG.AppQueue = retFrame;
            Console.WriteLine("[ModNS.IPProc] Echoed IP frame back");

            return (int)ReturnVal.OK;
        }

        // Process LCP message
        protected int LCPProc(InteropFrame argFrame, ref ACEntry argAC)
        {
            L3np.LCP.LCPFrame sendFrame;
            L3np.LCP.LCPFrame lcpFrame = new L3np.LCP.LCPFrame();
            InteropFrame cacheFrame;
            int newSid;

            lcpFrame.byteArray = argFrame.payload;
            lcpFrame.Decode();

            switch (lcpFrame.code)
            {
                case (int)L3np.Codes.ConfigureRequest:
                    // Manage configuration request
                    myL3np.Lcp.ConfigureRequestServer(lcpFrame, out sendFrame,
out newSid);
                    argAC.sessionid = newSid;
                    cacheFrame = new InteropFrame(argAC.cid, argAC.bsid,
sendFrame.byteArray);
                    myIAG.AppQueue = cacheFrame;
                    break;

                default:
                    break;
            }

            return (int)ReturnVal.OK;
        }

        //  Process IPCP message
        protected int IPCPProc(InteropFrame argFrame, ref ACEntry argAC)
        {
            L3np.IPCP.IPCPFrame sendFrame;
            L3np.IPCP.IPCPFrame ipcpFrame = new L3np.IPCP.IPCPFrame();
            InteropFrame cacheFrame;

            ipcpFrame.byteArray = argFrame.payload;
            ipcpFrame.Decode();

            switch (ipcpFrame.code)
            {
                case (int)L3np.Codes.ConfigureRequest:
                    // Manage configuration request
                    myL3np.Ipcp.ConfigureRequestServer(ipcpFrame, out 
sendFrame);
                    cacheFrame = new InteropFrame(argAC.cid, argAC.bsid, 
sendFrame.byteArray);
                    myIAG.AppQueue = cacheFrame;
                    break;

                default:
                    break;
            }


            return (int)ReturnVal.OK;
        }

        // Not implimented
        protected int AppProc(NetBack.Data argData)
        {
            //
            return (int)ReturnVal.Error;
        }

        // Find an AC user by session id
        protected void FindAC(int argSid, out ACEntry outAC)
        {
            Console.WriteLine("[L3np.FindAC] Looking for session id: {0}", 
argSid);
            lock (userList)
            {
                outAC = userList.First(delegate(ACEntry e)
                {
                    Console.WriteLine("[L3np.FindAC] Checking entry...");
                    if (e.sessionid == argSid)
                    {
                        Console.WriteLine("Entry Matches!");
                        return true;
                    }
                    else
                    {
                        Console.WriteLine("Entry does not match.");
                        return false;
                    }
                });
            }
        }

        // Create a new AC entry
        protected int NewAC(int argBSID, int argCID, out ACEntry outAC)
        {
            outAC = new ACEntry();
            outAC.bsid = argBSID;
            outAC.cid = argCID;

            // Add to the list of users
            lock (userList)
            {
                userList.AddLast(outAC);
                Console.WriteLine("[ModNS.NewAC] Added new user. Total
users:{0}", userList.Count);
            }

            return (int)ReturnVal.OK;
        }

        // The property to allow ModIAG to access the incoming message queue.
        public InteropFrame NetQueue
        {
            get
            {
                InteropFrame tempData;
                lock (netQueue)
                {
                    tempData = netQueue.Dequeue();
                    Console.WriteLine("[ModNS.NetQueue.get] Removed from queue.
Total: {0}", netQueue.Count);

                }
                return tempData;
            }

            set
            {
                lock (netQueue)
                {
                    netQueue.Enqueue(value);
                    Console.WriteLine("[ModNS.NetQueue.set] Added to queue.
Total: {0}", netQueue.Count);
                }
            }
        }

        // Define the frame style used for communication between the IAG and the
NS modules
        public class InteropFrame
        {
            public int cid;
            public int bsid;
            public byte[] payload;

            public InteropFrame() {}
            public InteropFrame(int argCid, int argBsid, byte[] argPayload)
            {
                cid = argCid;
                bsid = argBsid;
                payload = argPayload;
            }
        }

        // Entry defining an user (AC client).
        public class ACEntry
        {
            public int sessionid;
            public int bsid;
            public int cid;
        }



    }
}
```

B.4.3 ModTun.cs

```
/* Name: ModTun.cs
 * Interfaces with the shared library to provide virtual network interface
 * capabilities and capture user traffic. Functions as a subprocess within
 * the SS node. Interfaces with ModAC to send/receive traffic
 *
 * Inspired by the thin C interface for tun/tap used by the Ipop project.
* C code in ipop/src/c-lib/linux_tap.c
* C# code in ipop/src/IpopNode/Ethernet.cs
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Threading;
using System.Runtime.InteropServices;

namespace WiMAXDSRC
{
    public class ModTun : ModBase
    {
        // The interface with our external Tun/Tap Module

        // Open virtual network interface
        [DllImport("libtuntap")]
        private static extern int TunOpen(byte[] argRetDev);

        // Close virtual network interface
        [DllImport("libtuntap")]
        private static extern void TunClose(int argFd);

        // Read traffic from virtual network interface
        [DllImport("libtuntap")]
        private static extern int TunRead(int argFd, byte[] argPacket, int
argLength);

        // Write traffic to virtual network interface
        [DllImport("libtuntap")]
        private static extern int TunWrite(int argFd, byte[] argPacket, int
argLength);

        // Send message to set IP address (not yet implimented)
        [DllImport("libtuntap")]
        private static extern int TunSetIP(string argDev, string argIP);

        // Queue of info from the AC
        protected Queue<byte[]> acQueue;
        ModAC myAC;

        // File descriptor
        int tunFd;
        string tunName = null;
        byte[] tunBuf = null;
        static int bufferLength = 1024;

        public ModTun(ModAC argAC)
        {
            myAC = argAC;
            acQueue = new Queue<byte[]>();
        }

        public override int Run()
        {
            Thread tunMain = new Thread(new ThreadStart(TunMain));
            tunMain.Start();

            return (int)ReturnVal.OK;
        }

        public void TunMain()
        {
            byte[] tempBuf = new byte[bufferLength];

            Console.WriteLine("[ModTun.TunMain] Started new thread");

            // Open new device
            tunFd = TunOpen(tempBuf);
            // Copy name to string
            tunName = System.Text.ASCIIEncoding.ASCII.GetString(tempBuf);
            Console.WriteLine("[ModTun.TunMain] Opened new device: {0}",
tunName);

            // Start up the read handler
            Thread tunRead = new Thread(new ThreadStart(StartRead));
            tunRead.Start();

            while (true)
            {
                // Check for outstanding info on the queue
                // and send it
                while (acQueue.Count > 0)
                {
                    Console.WriteLine("[ModTun.TunMain] Something to forward!",
tunName);
                    tempBuf = ACQueue;
                    Console.WriteLine("[ModTun.TunMain] Took something off the
queue", tunName);
```

```
                    Console.WriteLine("Sending tunFd:{0}, tempBuf.Length:{1}",
tunFd, tempBuf.Length);
                    TunWrite(tunFd, tempBuf, tempBuf.Length);
                    Console.WriteLine("[ModTun.TunMain] Forwarded data to {0}",
tunName);
                }

                Thread.Sleep(SleepTime);
            }

            // return (int)ReturnVal.OK;
        }

        // Start read handler from the shared library
        private void StartRead()
        {
            Console.WriteLine("[ModTun.StartRead] Started read handler");
            int readLen;
            tunBuf = new byte[bufferLength];
            byte[] sendBuf;

            while (true)
            {
                readLen = TunRead(tunFd, tunBuf, bufferLength);
                Console.WriteLine("[ModTun.StartRead] Read {0} bytes", readLen);

                // Put the received message on the AC's tunnel queue.
                sendBuf = new byte[readLen];
                Array.Copy(tunBuf, sendBuf, readLen);
                myAC.TunQueue = sendBuf;

            }
        }

        // Setting IP not yet implimented
        public int SetIP(string argIP)
        {
            return TunSetIP(tunName, argIP);
        }

        // The property to allow ModAC to access the incoming message queue
        public byte[] ACQueue
        {
            get
            {
                byte[] tempData;
                lock (acQueue)
                {
                    tempData = acQueue.Dequeue();
                    Console.WriteLine("[ModAC.ACQueue.get] Removed from queue.
Total: {0}", acQueue.Count);

                }
                return tempData;
            }
            set
            {
                lock (acQueue)
                {
                    acQueue.Enqueue(value);
                    Console.WriteLine("[ModAC.ACQueue.set] Added to queue.
Total: {0}", acQueue.Count);
                }
            }
        }
    }
}
```

B.5 Network Protocol Base Class

B.5.1 NetBase.cs

```
/* Name: NetBase.cs
 * The base class for all network protocols. This includes the basic
 * functions for opening TCP/IP network connections and
 * reading/writing data. The derived classes extend this functionality
 * specific to the given protocol. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;

namespace WiMAXDSRC
{
    public class NetBase
    {
        static int bufferLength = 1024;

        protected int mode;
        protected int type;
        bool listenFlag;

        protected Socket socket;
        protected NetworkStream stream;

        protected const int fieldSize = sizeof(int);
        public ASCIIEncoding stringEncoder = new ASCIIEncoding();

        // Basic structure for transmitting/receiving data
        public class Data
        {
            public int length;
            public byte[] payload;

            public Data() {}
            public Data(byte[] argPayload)
            {
                payload = argPayload;
                length = payload.Length;
            }

        }

        /* The different roles that a given connection
         * can take */

        public enum netMode : int
        {
            Server,
            Client,
            Relay
        }

        /* Specific module types */
        public enum netType : int
        {
            Unknown,
            // Wimax types
            SS,
            RS,
            BS,
            IAG,
            // L3np types
            AC,
            NS
        }

        // Constructor
        public NetBase(netMode argMode, netType argType) : this(argMode,
argType, null) { }
        public NetBase(netMode argMode, netType argType, Socket argSocket)
        {
            mode = (int)argMode;
            type = (int)argType;

            socket = argSocket;

            // If we pass a null socket, this is going to fail.
            try
            {
                stream = new NetworkStream(socket);
            }
            catch
            {
                stream = null;
            }
        }

        // Connect to another module (as a client)
        public int Connect(string argIP, int argPort)
        {
            try
            {
                socket = new Socket(AddressFamily.InterNetwork, 
SocketType.Stream, ProtocolType.Tcp);
                socket.Connect(IPAddress.Parse(argIP), argPort);

                stream = new NetworkStream(socket);
            }
            catch (Exception e)
            {
                Console.WriteLine("[NetBase.Connect] Failed to connect.");
                Console.WriteLine("Exception: {0}", e.Message);
                return -1;
            }
            Console.WriteLine("[NetBase.Connect] Connected to {0}:{1}", argIP, 
argPort);
            return 0;
        }

        // Set up a listener for incoming connections (as a server)
        public int Listen(int argPort, ParameterizedThreadStart argPTS)
        {
            Socket acceptSocket;
            Thread acceptThread;
            IPEndPoint acceptEP;

            try
            {
                IPAddress localAddress = IPAddress.Parse("127.0.0.1");
                socket = new Socket(AddressFamily.InterNetwork, 
SocketType.Stream, ProtocolType.Tcp);

                socket.Bind(new IPEndPoint(IPAddress.Any, argPort));
                socket.Listen(10);
            }
            catch
            {
                Console.WriteLine("[NetBase.Listen] Failed to bind to port {0}", 
argPort);
                return -1;
            }

            Console.WriteLine("[NetBase.Listening] Listening for connections on 
{0}", argPort);
            listenFlag = true;

            while (listenFlag)
            {
                acceptSocket = socket.Accept();
                acceptEP = (IPEndPoint) acceptSocket.RemoteEndPoint;

                Console.WriteLine("[NetBase.Listening] Accepted new connection 
from {0}:{1}",
                    acceptEP.Address, acceptEP.Port);
                acceptThread = new Thread(argPTS);
                acceptThread.Start(acceptSocket);

            }
            return 0;
        }

        /* Handle the actual TCP/IP send. The derived classes extend
         * this further */
        public virtual int Send(Data argData)
        {
            try
            {
                stream.Write(argData.payload, 0, argData.length);
            }
            catch (Exception e)
            {
                Console.WriteLine("[NetBase.Send] Caught Exception\n{0}",
e.Message);
                return -1;
            }
            return 0;
        }

        /* Handle the actual TCP/IP read handler. The derived classes extend
         * this further */
        public virtual int AsyncReadStart(EntryBase argEntry, AsyncCallback
argCallback)
        {
            argEntry.readBuf = new byte[bufferLength];
            stream.BeginRead(argEntry.readBuf, 0, bufferLength, argCallback,
argEntry);

            return (int)ReturnVal.OK;
        }

        /* Handle the actual TCP/IP non-asynchronous read. The derived
         * classes extend this further */
        public virtual int Read(Data argData)
        {
            byte[] readBuffer = new byte[bufferLength];

            try
            {
                argData.length = stream.Read(readBuffer, 0, bufferLength);
                Console.WriteLine("[NetBase.Read] Read {0} bytes",
argData.length);
                argData.payload = new byte[argData.length];

                Array.Copy(readBuffer, argData.payload, argData.length);

            }
            catch (Exception e)
            {
                Console.WriteLine("[NetBase.Read] Caught Exception\n{0}",
e.Message);
                return -1;
            }

            return 0;
        }

        /* Handle the actual TCP/IP asynchronous read. The derived
         * classes extend this further */
        public int AsyncRead(Data argData, EntryBase argEntry, IAsyncResult ar)
        {
            argData.length = stream.EndRead(ar);

            argData.payload = new byte[argData.length];
            Array.Copy(argEntry.readBuf, argData.payload, argData.length);

            return (int)ReturnVal.OK;
        }
    }
}
```

B.6 WiMAX Network Protocol

B.6.1 Wimax.cs

```
/* Name: Wimax.cs
 * The implimentation of WiMAX for communication both between
 * the BS and RS nodes (Wimax over DSRC), as well as between
 * the RS nodes and the BS node (uplink using 16j tunneling).
 *
 * This is the main file */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public partial class Wimax : NetBase
    {
        // Constructors
        public Wimax(netMode argMode, netType argType, EntryBase argEntry) :
base(argMode, argType, null)
        {
            entry = argEntry;
        }

        public Wimax(netMode argMode, netType argType, EntryBase argEntry,
Socket argSocket)
            : base(argMode, argType, argSocket)
        {
            entry = argEntry;
```

```
}

// Handle non-asychronous message sending.
public int Send(Frame argFrame)
{
    Data sendData;
    SendEncode(argFrame, out sendData);
    return Send(sendData);
}

// Queue messages for sending at the appropriate time
public int AsyncSend(Frame argFrame)
{
    return AsyncSend(argFrame, entry);
}

public int AsyncSend(Frame argFrame, EntryBase argEntry)
{
    Data sendData;
    SendEncode(argFrame, out sendData);

    lock (argEntry.sendQueue)
    {
        argEntry.sendQueue.Enqueue(sendData);
    }

    return (int)ReturnVal.OK;
}

// Queue a message to be relayed (tunneled)
public int RelayQueue(Frame argFrame, EntryRS argRS)
{
    Data sendData;
    SendEncode(argFrame, out sendData);

    lock (argRS.relayQueue)
    {
        argRS.relayQueue.Enqueue(sendData);
    }

    return (int)ReturnVal.OK;
}

/* Combine all the messages in the relay queue
 * into a single tunneled message for TX to
 * the BS node */
public int RelaySend(EntryRS argRS)
{
    Data[] messages;
    byte[] tunPayload;

    TunFrame relayFrame;
    int count, c;
    int payloadLen = 0;
    int d = 0;
```

```
            lock (argRS.relayQueue)
            {
                // At this point, we're going to dequeue everything.
                // Realistically, we probably will have a limit to the
                // number of messages to take off.

                count = argRS.relayQueue.Count;
                messages = new Data[count];

                for (c = 0; c < count; c++)
                {
                    messages[c] = argRS.relayQueue.Dequeue();
                    payloadLen += messages[c].length;
                }
            }

            tunPayload = new byte[payloadLen];

            for (c = 0; c < count; c++)
            {
                Array.Copy(messages[c].payload, 0, tunPayload, d,
messages[c].length);
                d += messages[c].length;
            }

            relayFrame = new TunFrame(argRS.mtCid, count, tunPayload);
            Console.WriteLine("[Wimax.RelaySend] Relayed {0} messages", count);

            return Send(relayFrame);
        }

        /* Encode a WiMAX frame into a byte stream for transmission */
        public int SendEncode(Frame argFrame, out Data argData)
        {
            int c = 0;
            byte[] sendBuffer;

            if (argFrame is MgmtFrame)
            {
                argFrame.length = (3 * fieldSize) + argFrame.payload.Length;
            }
            else if (argFrame is TunFrame)
            {
                argFrame.length = (3 * fieldSize) + argFrame.payload.Length;
            }
            else
            {
                argFrame.length = (2 * fieldSize) + argFrame.payload.Length;
            }

            sendBuffer = new byte[argFrame.length];

            System.BitConverter.GetBytes(argFrame.cid).CopyTo(sendBuffer, 0);
            c += fieldSize;
            System.BitConverter.GetBytes(argFrame.length).CopyTo(sendBuffer, c);
            c += fieldSize;

            if (argFrame is MgmtFrame)
            {

System.BitConverter.GetBytes(((MgmtFrame)argFrame).mgmtMsg).CopyTo(sendBuffer,
c);
                c += fieldSize;
            }
            else if (argFrame is TunFrame)
            {

System.BitConverter.GetBytes(((TunFrame)argFrame).tunEntries).CopyTo(sendBuffer,
c);
                c += fieldSize;
            }

            argFrame.payload.CopyTo(sendBuffer, c);
            argData = new Data(sendBuffer);

            return (int)ReturnVal.OK;
        }

        // Read a message from the WiMAX connection
        public int Read(out Frame argFrame)
        {
            NetBase.Data readData = new NetBase.Data();
            if (Read(readData) == (int)ReturnVal.OK)
            {
                return ReadDecode(out argFrame, readData.payload, 0);
            }
            else
            {
                argFrame = new Frame();
                return (int)ReturnVal.Fail;
            }
        }

        // Perform an asyncronous read of WiMAX traffic
        public int AsyncRead(Data argData, IAsyncResult ar, out Frame argFrame)
        {
            AsyncRead(argData, entry, ar);
            return ReadDecode(out argFrame, argData.payload, 0);
        }

        // Decode a relay message - break back into the individual messages
        public int RelayRead(TunFrame argFrame, out Frame[] outFrames)
        {
            if (!(entry is EntryRS) && !(entry is EntryBS))
            {
                throw new Exception("[Wimax.RelayRead] Can only be run by RS or
BS");
            }
            int c;
            int d = 0;

            outFrames = new Frame[argFrame.tunEntries];

            for (c = 0; c < argFrame.tunEntries; c++)
            {
                d = RelayDecode(out outFrames[c], argFrame.payload, d);
            }

            return (int)ReturnVal.OK;
        }

        // Find the SS node with the specified CID
        public void RelayFindSS(int argCid, LinkedList<EntrySS> argList, out
EntrySS outSS)
        {
            lock (argList)
            {
                outSS = argList.First(delegate(EntrySS s)
                {
                    // Check each node for its "type" against the CID
                    // If it's invalid, then we don't have a match
                    if (TypeDecode(s, argCid) == MsgType.Invalid)
                    {
                        return false;
                    }
                    else
                    {
                        return true;
                    }
                });

            }
        }

        // Find a relay station with the specific CID
        public void RelayFindRS(int argCid, LinkedList<EntryRS> argList, out
EntryRS outRS)
        {
            lock (argList)
            {
                lock (argList)
                {
                    outRS = argList.First(delegate(EntryRS s)
                    {
                        if (TypeDecode(s, argCid) == MsgType.Invalid)
                        {
                            return false;
                        }
                        else
                        {
                            return true;
                        }
                    });
                }
            }
        }

        // Search for a user with the specified ranging code
        public void RelayFindRanging(Frame argFrame, LinkedList<EntrySS>
argList, out EntrySS outSS)
        {

            // This command ONLY works on management mesages of type RNG_RSP
            MgmtFrame myFrame = new MgmtFrame();
            myFrame.ConvertFrame(argFrame);

            if(myFrame.mgmtMsg != (int)mgmtType.RNG_RSP)
            {
                outSS = null;
                return;
            }

            RNG_RSP rngRsp = new RNG_RSP(entry);
            rngRsp.byteArray = myFrame.payload;
            rngRsp.Decode();

            Console.WriteLine("[Wimax.RelayFindRanging] Looking for ranging
code: {0}", rngRsp.rangingCode);
            lock (argList)
            {
                outSS = argList.First(delegate(EntrySS s)
                {
                    Console.WriteLine("[Wimax.RelayFindRanging] Checking
entry...");
                    if (s.rangingCode == rngRsp.rangingCode)
                    {
                        Console.WriteLine("Entry Matches!");
                        return true;
                    }
                    else
                    {
                        Console.WriteLine("Entry does not match.");
                        return false;
                    }
                });
            }
        }

        // Decode a relay frame
        protected int RelayDecode(out Frame outFrame, byte[] inArray, int
argIndex)
        {
            int cid, length, index;

            cid = length = index = 0;
            index = ReadHeader(inArray, argIndex, ref cid, ref length);
            // Decode as normal frames
            return ReadConstruct(out outFrame, inArray, index, cid, length,
MsgType.Data);
        }

        // Reconstruct a received message into a frame structure
        protected int ReadDecode(out Frame outFrame, byte[] inArray, int
argIndex)
        {
            int cid, length, index;
            MsgType type;

            cid = length = index = 0;
            index = ReadHeader(inArray, argIndex, ref cid, ref length);
            type = TypeDecode(entry, cid);
            return ReadConstruct(out outFrame, inArray, index, cid, length,
type);
        }

        // Read the generic WiMAX header to find CID and Length information
        protected int ReadHeader(byte[] argArray, int argIndex, ref int argCid,
ref int argLength)
        {
            int c = argIndex;

            argCid = System.BitConverter.ToInt32(argArray, c);
            c += fieldSize;

            argLength = System.BitConverter.ToInt32(argArray, c);
            c += fieldSize;

            return c;
        }

        // Actually perform the reconstruction of the WiMAX message.
        protected int ReadConstruct (out Frame argFrame, byte[] argArray, int
argIndex, int argCid,
            int argLength, MsgType argType)
        {
            int payloadLength;
            int c = argIndex;

            // The reconstruction process varies depending on message type
            if (argType == MsgType.Management)
            {
                argFrame = new MgmtFrame();

                ((MgmtFrame)argFrame).mgmtMsg =
System.BitConverter.ToInt32(argArray, c);
                c += fieldSize;

                payloadLength = argLength - (3 * fieldSize);
            }
            else if (argType == MsgType.Data)
            {
                argFrame = new Frame();
                payloadLength = argLength - (2 * fieldSize);
            }
            else if (argType == MsgType.Tunnel)
            {
                // Handle a tunnel
                argFrame = new TunFrame();

                ((TunFrame)argFrame).tunEntries =
System.BitConverter.ToInt32(argArray, c);
                c += fieldSize;

                payloadLength = argLength - (3 * fieldSize);
            }
            else
            {
                // invalid message. Drop
                argFrame = new Frame();
                return (int)ReturnVal.Error;
            }

            argFrame.cid = argCid;
            argFrame.length = argLength;

            argFrame.payload = new byte[payloadLength];
            Array.Copy(argArray, c, argFrame.payload, 0, payloadLength);
            c += payloadLength;

            return c;
        }

        // Decode message type
        private MsgType TypeDecode(EntryBase argEntry, int argCid)
        {
            /* Based on the CID, we classify the message into one of 4 types
             * 1. Management message (default, basic and primary cid)
             * 2. User data message (managed CID)
             * 3. Tunnelled traffic (tunnel/management tunnel CID)
             * 4. Invalid message (any other CID)
             *
             * Note: CIDs only have meaning for BS and RS node types.
             * Everything else returns as invalid.
             */

            if (argEntry is EntrySS)
            {
                EntrySS mySS = (EntrySS) argEntry;

                if (argCid == defaultCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == mySS.basicCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == mySS.primaryCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == mySS.transportCid)
                {
                    return MsgType.Data;
                }
                else
                {
                    return MsgType.Invalid;
                }
            }
            else if (argEntry is EntryRS)
            {
                EntryRS myRS = (EntryRS)argEntry;

                if (argCid == defaultCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == myRS.basicCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == myRS.primaryCid)
                {
                    return MsgType.Management;
                }
                else if (argCid == myRS.tCid)
                {
                    return MsgType.Tunnel;
                }
                else if (argCid == myRS.mtCid)
                {
                    return MsgType.Tunnel;
                }
                else
                {
                    return MsgType.Invalid;
                }
            }
            else if (argEntry is EntryBS)
            {
                EntryBS myBS = (EntryBS)argEntry;

                // BS can handle default CID messages
                // for initial ranging
                if (argCid == defaultCid)
                {
                    return MsgType.Management;
                }
                else
                {
```

```
                    return MsgType.Invalid;
                }
            }
            else
            {
                // CIDs are meaningless outside of this context
                return MsgType.Invalid;
            }
        }

        // Generate an appropriate ranging code
        public int GetRCode()
        {
            Random random = new Random();
            // Return 8-bit number
            return random.Next(1, 255);
        }

        public int GetDSXTransID()
        {
            Random random = new Random();
            // Return 16-bit number
            return random.Next(0, 65535);
        }
    }
}
```

B.6.2 Wimax.definitions.cs

```
/* Name: Wimax.definitions.cs
 * The implimentation of WiMAX for communication both between
 * the BS and RS nodes (Wimax over DSRC), as well as between
 * the RS nodes and the BS node (uplink using 16j tunneling).
 *
 * This file stores definitions of the WiMAX objects.
 * This is primarily frame and message structures. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class Wimax : NetBase
    {
        protected EntryBase entry;

        // The generic definition of a WiMAX frame
        public class Frame
        {
            public int cid;
            public int length;
            public bool relay;

            public byte[] payload;
            public Frame() { }
            public Frame(int argCid, byte[] argPayload)
            {
                cid = argCid;
                payload = argPayload;
                length = (2 * fieldSize) + payload.Length;
            }

            public int SetFrame(int argCid, byte[] argPayload)
            {
                cid = argCid;
                payload = argPayload;
                length = (2 * fieldSize) + payload.Length;
                return (int)ReturnVal.OK;
            }
        }

        // Specific definition of a WiMAX Management frame
        public class MgmtFrame : Wimax.Frame
        {
            public int mgmtMsg;

            public MgmtFrame() : base() { }
            public MgmtFrame(int argCid, int argMsg, byte[] argPayload)
                : base(argCid, argPayload)
            {
                mgmtMsg = argMsg;
                length = (3 * fieldSize) + argPayload.Length;
            }

            public int SetFrame(int argCid, int argMsg, byte[] argPayload)
            {
                SetFrame(argCid, argPayload);
                mgmtMsg = argMsg;
                length = (3 * fieldSize) + argPayload.Length;
                return (int)ReturnVal.OK;
            }

            // Convert a generic frame structure into a WiMAX frame
            public int ConvertFrame(Frame argFrame)
            {
                // Take out the management message
                mgmtMsg = System.BitConverter.ToInt32(argFrame.payload, 0);

                // Modify the payload
                int payloadLen = argFrame.length - (3 * fieldSize);
                byte[] newPayload = new byte[payloadLen];
                Array.Copy(argFrame.payload, fieldSize, newPayload, 0,
payloadLen);

                // Set the remaining parameters
                SetFrame(argFrame.cid, newPayload);
                length = (3 * fieldSize) + newPayload.Length;
                return (int)ReturnVal.OK;
            }
        }

        // Specific definition of a WiMAX Tunnel frame
        public class TunFrame : Wimax.Frame
        {
            public int tunEntries;

            public TunFrame() : base() { }
            public TunFrame(int argCid, int argEntries, byte[] argPayload)
                : base(argCid, argPayload)
            {
                tunEntries = argEntries;
                length = (3 * fieldSize) + argPayload.Length;
            }

            public int SetFrame(int argCid, int argEntries, byte[] argPayload)
            {
                SetFrame(argCid, argPayload);
                tunEntries = argEntries;
                length = (3 * fieldSize) + argPayload.Length;
                return (int)ReturnVal.OK;
            }
        }

        // Management message types supported by the system
        public enum mgmtType : int
        {
            RNG_REQ = 4,
            RNG_RSP = 5,

            REG_REQ = 6,
            REG_RSP = 7,

            DSA_REQ = 11,
            DSA_RSP = 12,
            DSA_ACK = 13,

            DSX_RVD = 30,

            SYNC = 255
        }

        // Message types supported by the system
        protected enum MsgType : int
        {
            Management,
            Data,
            Tunnel,
            Invalid
        }

        public const int defaultCid = 0;
```

```
// Generic definition of a WiMAX management message
// (payload for a management frame)
abstract public class Message
{
    public byte[] byteArray;
    protected int fields;
    protected int size;
    protected EntryBase entry;

    abstract public int Encode();
    abstract public int Decode();

    public int EncodePacket(params int[] argVals)
    {
        byteArray = new byte[size];
        int c = 0;

        foreach (int i in argVals)
        {
            (System.BitConverter.GetBytes(i)).CopyTo(byteArray, c);
            c += fieldSize;
        }

        return (int)ReturnVal.OK;
    }

    public int DecodePacket(out int[] argVals)
    {
        int c, d = 0;
        argVals = new int[fields];

        for (c = 0; c < fields; c++)
        {
            argVals[c] = System.BitConverter.ToInt32(byteArray, d);
            d += fieldSize;
        }

        return (int)ReturnVal.OK;
    }
}

public class RNG_REQ : Message
{
    public int rangingCode;
    public int macVersion;
    public int msRanging;
    public int nodeType;

    public RNG_REQ(EntryBase argEntry)
    {
        fields = 3;
        size = fields * fieldSize;
    }

    public override int Encode()
```

```
            {
                return EncodePacket(rangingCode, macVersion, msRanging);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                rangingCode = fields[0];
                macVersion = fields[1];
                msRanging = fields[2];

                return retVal;
            }
        }

        public class RNG_RSP : Message
        {
            public int rangingCode;
            public int macVersion;
            public int regBSID;
            public int basicCID;
            public int primaryCID;

            public RNG_RSP(EntryBase argEntry)
            {
                fields = 5;
                size = fields * fieldSize;
            }

            public override int Encode()
            {
                return EncodePacket(rangingCode, macVersion, regBSID, basicCID,
primaryCID);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                rangingCode = fields[0];
                macVersion = fields[1];
                regBSID = fields[2];
                basicCID = fields[3];
                primaryCID = fields[4];

                return retVal;
            }
        }

        public class REG_REQ : Message
        {
            public int uplinkCidSupport;
            public int ssMgmtSupport;
            public int ipMgmtMode;

            public REG_REQ(EntryBase argEntry)
            {
                entry = argEntry;
                fields = 3;
                size = fields * fieldSize;
            }

            public override int Encode()
            {
                return EncodePacket(uplinkCidSupport, ssMgmtSupport,
ipMgmtMode);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                uplinkCidSupport = fields[0];
                ssMgmtSupport = fields[1];
                ipMgmtMode = fields[2];

                return retVal;
            }
        }

        public class REG_RSP : Message
        {
            public int response;
            public int ssMgmtSupport;

            // The SM CID would be assigned here, but we're not using it

            // for RS only
            public int gwBsid;
            public int tCid;
            public int mtCid;

            public REG_RSP(EntryBase argEntry)
            {
                // The value depends on if this is a RS node or not
                entry = argEntry;

                if (entry is EntryRS)
                {
                    fields = 5;
                }
                else
                {
                    fields = 2;
                }
                size = fields * fieldSize;
            }

            public override int Encode()
            {
                if (entry is EntryRS)
                {
                    return EncodePacket(response, ssMgmtSupport, gwBsid, tCid,
mtCid);
                }
                else
                {
                    return EncodePacket(response, ssMgmtSupport);
                }
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                response = fields[0];
                ssMgmtSupport = fields[1];


                if (entry is EntryRS)
                {
                    gwBsid = fields[2];
                    tCid = fields[3];
                    mtCid = fields[4];
                }

                return retVal;
            }
        }
        // Confirmation codes
        public enum CCValues : int
        {
            OK = 0,
            Reject_Other = 1
        }

        public class DSA_REQ : Message
        {
            public int transactionId;


            public DSA_REQ(EntryBase argEntry)
            {
                fields = 1;
                size = fields * fieldSize;
            }
```

```
    public override int Encode()
    {
        return EncodePacket(transactionId);
    }

    public override int Decode()
    {
        int[] fields;

        int retVal = DecodePacket(out fields);

        transactionId = fields[0];

        return retVal;
    }
}

public class DSA_RSP : Message
{
    public int transactionId;
    public int confirmationCode;
    public int sfid;
    public int cid;


    public DSA_RSP(EntryBase argEntry)
    {
        fields = 4;
        size = fields * fieldSize;
    }

    public override int Encode()
    {
        return EncodePacket(transactionId, confirmationCode, sfid, cid);
    }

    public override int Decode()
    {
        int[] fields;

        int retVal = DecodePacket(out fields);

        transactionId = fields[0];
        confirmationCode = fields[1];
        sfid = fields[2];
        cid = fields[3];
        return retVal;
    }
}

public class DSA_ACK : Message
{
    public int transactionId;
    public int confirmationCode;
    public DSA_ACK(EntryBase argEntry)
    {
        fields = 2;
        size = fields * fieldSize;
    }

    public override int Encode()
    {
        return EncodePacket(transactionId, confirmationCode);
    }

    public override int Decode()
    {
        int[] fields;

        int retVal = DecodePacket(out fields);

        transactionId = fields[0];
        confirmationCode = fields[1];

        return retVal;
    }
}

// DSx Received Message
public class DSX_RVD : Message
{
    public int transactionId;
    public int confirmationCode;

    public DSX_RVD(EntryBase argEntry)
    {
        fields = 2;
        size = fields * fieldSize;
    }

    public override int Encode()
    {
        return EncodePacket(transactionId, confirmationCode);
    }

    public override int Decode()
    {
        int[] fields;

        int retVal = DecodePacket(out fields);

        transactionId = fields[0];
        confirmationCode = fields[1];

        return retVal;
    }
}
```

```
        /* SYNC message was specially created for the
         * WiMAX over DSRC system, due to our lack of
         * a physical air interface */
        public class SYNC : Message
        {
            public int version;
            public int bsid;

            public SYNC(EntryBase argEntry)
            {
                fields = 2;
                size = fields * fieldSize;
            }

            public override int Encode()
            {
                return EncodePacket(version, bsid);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                version = fields[0];
                bsid = fields[1];

                return retVal;
            }
        }
    }
}
```

B.6.3 Wimax.handlers.cs

```
/* Name: Wimax.handlers.cs
 * The implimentation of WiMAX for communication both between
 * the BS and RS nodes (Wimax over DSRC), as well as between
 * the RS nodes and the BS node (uplink using 16j tunneling).
 *
 * This file stores event handlers for the WiMAX communications.
 * These are called by the message handlers in the modules. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class Wimax : NetBase
    {

        // The client, server, and relay handler interfaces for SYNC messages
        public int SYNCClient(MgmtFrame inFrame, out MgmtFrame outFrame)
        {
            SYNC sync = new SYNC(entry);
            SYNCHandler(ref inFrame, ref sync, null, null);

            // Prepare response
            RNG_REQ rngReq = new RNG_REQ(entry);
            outFrame = new MgmtFrame();
            return RNG_REQHandler(ref outFrame, ref rngReq, null, null);
        }

        public int SYNCServer(out MgmtFrame outFrame, EntryBS argBS)
        {
            SYNC sync = new SYNC(entry);
            outFrame = new MgmtFrame();

            return SYNCHandler(ref outFrame, ref sync, argBS, null);
        }

        public int SYNCRelay(out MgmtFrame outFrame, EntryRS argRS)
        {

            SYNC sync = new SYNC(entry);
            outFrame = new MgmtFrame();

            return SYNCHandler(ref outFrame, ref sync, null, argRS);
        }

        // The actual SYNC handler
        protected int SYNCHandler(ref MgmtFrame argFrame, ref SYNC sync, EntryBS
argBS, EntryRS argRS)
        {

            if (mode == (int)Wimax.netMode.Client)
            {
                sync.byteArray = argFrame.payload;
                sync.Decode();

                Console.WriteLine("[Wimax.SYNCHandler] Read version:{0},
rsBSID:{1}", sync.version,
                    sync.bsid);

                if (entry is EntrySS)
                {
                    ((EntrySS)entry).servingBSID = sync.bsid;
                    ((EntrySS)entry).version = sync.version;
                    ((EntrySS)entry).status = (int)Status.Synced;
                }
                else if (entry is EntryRS)
                {
                    ((EntryRS)entry).servingBSID = sync.bsid;
                    ((EntryRS)entry).version = sync.version;
                    ((EntryRS)entry).status = (int)Status.Synced;
                }
                else
                {
                    throw new Exception("SYNCHandler client only works for types
EntrySS and EntryRS");
                    //return (int)ReturnVal.Error;
                }

                entry.status = (int)Status.Synced;

                return (int)ReturnVal.OK;

            }
            else if ((mode == (int)Wimax.netMode.Server) || (mode ==
(int)Wimax.netMode.Relay))
            {
                sync.version = Program.programVersion;

                if (mode == (int)Wimax.netMode.Relay)
                {
                    sync.bsid = argRS.myBSID;

                }
                else if (mode == (int)Wimax.netMode.Server)
                {
                    sync.bsid = argBS.bsid;
                }
                else
                {
                    throw new Exception("SYNCHandler server only works for types
EntryRS and EntryBS");
                    //return (int)ReturnVal.Error;
                }

                Console.WriteLine("[Wimax.SYNCHandler] Sent version:{0},
rsBSID:{1}", sync.version,
                    sync.bsid);

                sync.Encode();
                argFrame.SetFrame(defaultCid, (int)Wimax.mgmtType.SYNC,
sync.byteArray);
                //AsyncSend(syncFrame);

                entry.status = (int)Status.Synced;

            }

            return (int)ReturnVal.OK;
        }

        // The server and relay handler interfaces for RNG_REQ
        public int RNG_REQServer(MgmtFrame inFrame, out MgmtFrame outFrame,
EntryBS argBS)
        {
            RNG_REQ rngReq = new RNG_REQ(entry);
            RNG_RSP rngRsp = new RNG_RSP(entry);
            outFrame = new MgmtFrame();
            RNG_REQHandler(ref inFrame, ref rngReq, argBS, null);

            rngRsp.rangingCode = rngReq.rangingCode;
            rngRsp.basicCID = argBS.myRes.GetCid((int)WimaxRes.CidType.Basic);
            rngRsp.primaryCID =
argBS.myRes.GetCid((int)WimaxRes.CidType.Primary);
            return RNG_RSPHandler(ref outFrame, ref rngRsp, argBS, null);

        }

        public int RNG_REQRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            // Copy some details and forward along

            RNG_REQ rngReq = new RNG_REQ(entry);

            return RNG_REQHandler(ref inFrame, ref rngReq, null, argRS);
        }

        // The actual RNG_REQ handler
        protected int RNG_REQHandler(ref MgmtFrame argFrame, ref RNG_REQ rngReq,
EntryBS argBS, EntryRS argRS)
        {
            if (mode == (int)Wimax.netMode.Client)
            {
                // Send RNG_REQ
                rngReq = new RNG_REQ(entry);
                rngReq.rangingCode = GetRCode();

                if (entry is EntryRS)
                {
                    ((EntryRS)entry).rangingCode = rngReq.rangingCode;
                }
                else if (entry is EntrySS)
                {
                    ((EntrySS)entry).rangingCode = rngReq.rangingCode;
                }

                rngReq.nodeType = type;
                rngReq.Encode();

                Console.WriteLine("[Wimax.RNG_REQHandler] Sent RNG_REQ, cid:{0},
ranging code:{1}, type:{2}",
                    argFrame.cid, rngReq.rangingCode, rngReq.nodeType);

                argFrame.SetFrame(defaultCid, (int)mgmtType.RNG_REQ,
rngReq.byteArray);
                //AsyncSend(reqFrame);
            }
            else if (mode == (int)Wimax.netMode.Server)
            {
                rngReq = new RNG_REQ(entry);
                rngReq.byteArray = argFrame.payload;
                rngReq.Decode();
                Console.WriteLine("[Wimax.RNG_REQHandler] Received RNG_REQ,
cid:{0}, ranging code:{1}, type:{2}",
                    argFrame.cid, rngReq.rangingCode, rngReq.nodeType);

                // Now we know if its an RS or SS
                type = rngReq.nodeType;

            }

            /* The relay node copies the ranging code, then passes the message
             * along for relaying */
            else if (mode == (int)Wimax.netMode.Relay)
            {
                if (entry is EntrySS)
                {
                    rngReq.byteArray = argFrame.payload;
                    rngReq.Decode();

                    Console.WriteLine("[Wimax.RNG_REQHandler] Received RNG_REQ,
cid:{0}, ranging code:{1}, type:{2}",
                        argFrame.cid, rngReq.rangingCode, rngReq.nodeType);

                    // Store the node type and ranging code
                    type = rngReq.nodeType;
                    ((EntrySS)entry).rangingCode = rngReq.rangingCode;

                    Console.WriteLine("Stored data and forwarded to BS");
                }
            }

            return (int)ReturnVal.OK;
        }

        // Client and relay handler interfaces for RNG_RSP
        public int RNG_RSPClient(MgmtFrame inFrame, out MgmtFrame outFrame)
        {
            RNG_RSP rngRsp = new RNG_RSP(entry);
            REG_REQ regReq = new REG_REQ(entry);
            outFrame = new MgmtFrame();

            RNG_RSPHandler(ref inFrame, ref rngRsp, null, null);
            return REG_REQHandler(ref outFrame, ref regReq, null, null);
        }

        public int RNG_RSPRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            RNG_RSP rngRsp = new RNG_RSP(entry);
            return RNG_RSPHandler(ref inFrame, ref rngRsp, null, argRS);
        }

        // The actual RNG_RSP handler
        protected int RNG_RSPHandler(ref MgmtFrame argFrame, ref RNG_RSP rngRsp,
EntryBS argBS, EntryRS argRS)
        {
            if (mode == (int)Wimax.netMode.Server)
            {

                // Cache our local copies

                if (entry is EntrySS)
                {
                    ((EntrySS)entry).basicCid = rngRsp.basicCID;
                    ((EntrySS)entry).primaryCid = rngRsp.primaryCID;
                }
                else if (entry is EntryRS)
                {
                    ((EntryRS)entry).basicCid = rngRsp.basicCID;
                    ((EntryRS)entry).primaryCid = rngRsp.primaryCID;
                }

                rngRsp.Encode();

                Console.WriteLine("[Wimax.RNG_RSPHandler] Sent RNG_RSP, cid:{0}, 
ranging code:{1}\nbasic cid:{2}, primary cid:{3}",
                    argFrame.cid, rngRsp.rangingCode, rngRsp.basicCID, 
rngRsp.primaryCID);

                argFrame.SetFrame(argFrame.cid, (int)Wimax.mgmtType.RNG_RSP, 
rngRsp.byteArray);
            }
            else if (mode == (int)Wimax.netMode.Client)
            {
                rngRsp.byteArray = argFrame.payload;
                rngRsp.Decode();

                Console.WriteLine("[Wimax.RNG_RSPHandler] Received RNG_RSP, 
cid:{0}, ranging code:{1}\nbasic cid:{2}, primary cid:{3}",
                    argFrame.cid, rngRsp.rangingCode, rngRsp.basicCID, 
rngRsp.primaryCID);

                if (entry is EntryRS)
                {
                    ((EntryRS)entry).basicCid = rngRsp.basicCID;
                    ((EntryRS)entry).primaryCid = rngRsp.primaryCID;
                    // Remove the ranging code
                    ((EntryRS)entry).rangingCode = 0;
                }
                else if (entry is EntrySS)
                {
                    ((EntrySS)entry).basicCid = rngRsp.basicCID;
                    ((EntrySS)entry).primaryCid = rngRsp.primaryCID;
                    // Remove the ranging code
                    ((EntrySS)entry).rangingCode = 0;
                }
                else
                {
                    throw new Exception("RNG_RSPHandler client only works for 
types EntryRS and EntrySS");
                    //return (int)ReturnVal.Error;
                }

                // Ranged
                entry.status = (int)Status.Ranged;
            }
            else if (mode == (int)Wimax.netMode.Relay)
            {
                // Copy the CIDs from the reply message
                rngRsp.byteArray = argFrame.payload;
                rngRsp.Decode();


                Console.WriteLine("[Wimax.RNG_RSPHandler] Received RNG_RSP to 
relay , cid:{0}, ranging code:{1}\nbasic cid:{2}, primary cid:{3}",
                    argFrame.cid, rngRsp.rangingCode, rngRsp.basicCID, 
rngRsp.primaryCID);

                if (entry is EntrySS)
                {
                    ((EntrySS)entry).basicCid = rngRsp.basicCID;
                    ((EntrySS)entry).primaryCid = rngRsp.primaryCID;
                    // Remove the ranging code
                    ((EntrySS)entry).rangingCode = 0;
                }
            }

            return (int)ReturnVal.OK;
        }

        /* The demonstration system is skipping the Authentication phase for now 
*/

        // REG_REQ handler interface for the server
        public int REG_REQServer(MgmtFrame inFrame, out MgmtFrame outFrame, 
EntryBS argBS)
        {
            REG_REQ regReq = new REG_REQ(entry);
            REG_RSP regRsp = new REG_RSP(entry);
            outFrame = new MgmtFrame();

            REG_REQHandler(ref inFrame, ref regReq, argBS, null);
            outFrame.cid = inFrame.cid;
            return REG_RSPHandler(ref outFrame, ref regRsp, argBS, null);

        }

        public int REG_REQRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            // No useful information here.
            return (int)ReturnVal.OK;
        }

        // The actual REG_REQ handler
        protected int REG_REQHandler(ref MgmtFrame argFrame, ref REG_REQ regReq, 
EntryBS argBS, EntryRS argRS)
```

```
        {
            if (mode == (int)Wimax.netMode.Client)
            {
                regReq = new REG_REQ(entry);

                // Unmanaged connection
                regReq.ipMgmtMode = 0;
                regReq.ssMgmtSupport = 0;

                // We support 3 CIDs - basic, managed, and transport
                regReq.uplinkCidSupport = 3;
                regReq.Encode();

                if(entry is EntrySS)
                {
                    argFrame.SetFrame(((EntrySS)entry).primaryCid,
(int)mgmtType.REG_REQ, regReq.byteArray);
                    //AsyncSend(reqFrame);
                }
                else if (entry is EntryRS)
                {
                    argFrame.SetFrame(((EntryRS)entry).primaryCid,
(int)mgmtType.REG_REQ, regReq.byteArray);
                    //AsyncSend(reqFrame);
                }

                Console.WriteLine("[Wimax.REG_REQHandler] Sent REG_REQ,
cid:{0}",
                   argFrame.cid);
            }
            else if (mode == (int)Wimax.netMode.Server)
            {
                regReq.byteArray = argFrame.payload;
                regReq.Decode();

                Console.WriteLine("[Wimax.REG_REQHandler] Received REG_REQ,
cid:{0}",
                    argFrame.cid);

            }
            else if (mode == (int)Wimax.netMode.Relay)
            {
                //
            }
            return (int)ReturnVal.OK;
        }

        // Version 1 for RS nodes
        public int REG_RSPClient(MgmtFrame inFrame)
        {
            REG_RSP regRsp = new REG_RSP(entry);

            return REG_RSPHandler(ref inFrame, ref regRsp, null, null);
        }
        // Version 2 for SS nodes
        public int REG_RSPClient(MgmtFrame inFrame, out MgmtFrame outFrame)
        {
            REG_RSP regRsp = new REG_RSP(entry);
            DSA_REQ dsaReq = new DSA_REQ(entry);

            outFrame = new MgmtFrame();

            REG_RSPHandler(ref inFrame, ref regRsp, null, null);
            return DSA_REQHandler(ref outFrame, ref dsaReq, null, null);
        }

        public int REG_RSPRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            // nothing useful here
            return (int)ReturnVal.OK;
        }

        // The actual REG_RSP handler
        protected int REG_RSPHandler(ref MgmtFrame argFrame, ref REG_RSP regRsp,
EntryBS argBS, EntryRS argRS)
        {
            if (mode == (int)Wimax.netMode.Server)
            {
                // Send the reply

                regRsp.response = 0; // OK
                regRsp.ssMgmtSupport = 0; // Not going to support secondary
managed

                Console.WriteLine("[Wimax.REG_RSPHandler] Sent REG_RSP, cid:{0},
response:{1}",
                    argFrame.cid, regRsp.response);

                // Special stuff for RS nodes
                if (entry is EntryRS)
                {
                    regRsp.gwBsid = argBS.myRes.getBsid();
                    regRsp.tCid =
argBS.myRes.GetCid((int)WimaxRes.CidType.Tunnel);
                    regRsp.mtCid =
argBS.myRes.GetCid((int)WimaxRes.CidType.ManagedTunnel);

                    Console.WriteLine("gwBsid:{0}, tCid:{1}, mtCid:{2}",
regRsp.gwBsid, regRsp.tCid,
                        regRsp.mtCid);

                    // Cache our local copies
                    ((EntryRS)entry).myBSID = regRsp.gwBsid;
                    ((EntryRS)entry).tCid = regRsp.tCid;
                    ((EntryRS)entry).mtCid = regRsp.mtCid;
                }

                regRsp.Encode();
                argFrame.SetFrame(argFrame.cid, (int)mgmtType.REG_RSP,
regRsp.byteArray);
                //AsyncSend(rspFrame);
            }
            else if (mode == (int)Wimax.netMode.Client)
            {
                regRsp.byteArray = argFrame.payload;
                regRsp.Decode();

                Console.WriteLine("[Wimax.REG_RSPHandler] Received REG_RSP,
cid:{0}, response:{1}",
                    argFrame.cid, regRsp.response);

                if (entry is EntryRS)
                {
                    Console.WriteLine("gwBsid:{0}, tCid:{1}, mtCid:{2}",
regRsp.gwBsid, regRsp.tCid,
                        regRsp.mtCid);

                    ((EntryRS)entry).myBSID = regRsp.gwBsid;
                    ((EntryRS)entry).tCid = regRsp.tCid;
                    ((EntryRS)entry).mtCid = regRsp.mtCid;

                    entry.status = (int)Status.Running;
                }
                else
                {
                    entry.status = (int)Status.Registered;
                }
            }
            return (int)ReturnVal.OK;
        }

        // DSA_REQ handler interface for the server
        public int DSA_REQServer(MgmtFrame inFrame, out MgmtFrame outFrame,
EntryBS argBS)
        {
            outFrame = new MgmtFrame();
            DSA_REQ dsaReq = new DSA_REQ(entry);
            DSX_RVD dsxRcv = new DSX_RVD(entry);

            // Handle DSA request
            DSA_REQHandler(ref inFrame, ref dsaReq, argBS, null);
            outFrame.cid = inFrame.cid;

            // Acknowledge recept of DCx message
            DSX_RVDHandler(ref outFrame, ref dsxRcv, argBS, null);

            // DSA_RSP will be processed by a seperate handler
            // (once the DCX_RVD is sent)

            return (int)ReturnVal.OK;
        }

        public int DSA_REQRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            return (int)ReturnVal.OK;
        }

        // The DSA_REQ handler
        protected int DSA_REQHandler(ref MgmtFrame argFrame, ref DSA_REQ dsaReq,
EntryBS argBS, EntryRS argRS)
        {
            if (!(entry is EntrySS))
            {
                throw new Exception("Wimax.DSA_REQHandler: Entry must be
EntrySS");
            }
            EntrySS mySS = (EntrySS)entry;

            if (mode == (int)Wimax.netMode.Client)
            {
                // Set the transaction ID and store a copy
                dsaReq.transactionId = GetDSXTransID();
                mySS.transactionID = dsaReq.transactionId;

                Console.WriteLine("[Wimax.REG_REQHandler] Sending DSA_REQ:
cid:{0}, transactionId:{1}",
                    argFrame.cid, dsaReq.transactionId);

                dsaReq.Encode();
                argFrame.SetFrame(mySS.primaryCid, (int)mgmtType.DSA_REQ,
dsaReq.byteArray);
            }
            else if (mode == (int)Wimax.netMode.Server)
            {
                dsaReq.byteArray = argFrame.payload;
                dsaReq.Decode();

                // Store copy of transaction ID
                ((EntrySS)entry).transactionID = dsaReq.transactionId;

                Console.WriteLine("[Wimax.REG_REQHandler] Received DSA_REQ:
cid:{0}, transactionId:{1}",
                    argFrame.cid, dsaReq.transactionId);
            }
            else if (mode == (int)Wimax.netMode.Relay)
            {
                //
            }
            return (int)ReturnVal.OK;
        }

        // The DSA_RSP handler interfaces for the client, server and relay
        public int DSA_RSPServer(MgmtFrame inFrame, out MgmtFrame outFrame,
EntryBS argBS)
        {

            // Process DSA request and send reply
            outFrame = new MgmtFrame();
            DSA_RSP dsaRsp = new DSA_RSP(entry);
            outFrame.cid = inFrame.cid;
            return DSA_RSPHandler(ref outFrame, ref dsaRsp, argBS, null);
        }

        public int DSA_RSPClient(MgmtFrame inFrame, out MgmtFrame outFrame)
        {
            outFrame = new MgmtFrame();
            DSA_RSP dsaRsp = new DSA_RSP(entry);
            DSA_ACK dsaAck = new DSA_ACK(entry);

            DSA_RSPHandler(ref inFrame, ref dsaRsp, null, null);
            return DSA_ACKHandler(ref outFrame, ref dsaAck, null, null);
        }

        public int DSA_RSPRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            DSA_RSP dsaRsp = new DSA_RSP(entry);
            return DSA_RSPHandler(ref inFrame, ref dsaRsp, null, argRS);
        }

        // Actual DSA_RSP handler
        protected int DSA_RSPHandler(ref MgmtFrame argFrame, ref DSA_RSP dsaRsp,
EntryBS argBS, EntryRS argRS)
        {
            if (!(entry is EntrySS))
            {
                throw new Exception("Wimax.DSA_RSPHandler: Entry must be
EntrySS");
            }
            EntrySS mySS = (EntrySS) entry;

            if (mode == (int)Wimax.netMode.Server)
            {
                dsaRsp.transactionId = mySS.transactionID;
                dsaRsp.cid =
argBS.myRes.GetCid((int)WimaxRes.CidType.Transport);

                // For the time being, we'll use SFID = CID
                dsaRsp.sfid = dsaRsp.cid;

                // Store our copies of the entries
                mySS.transportCid = dsaRsp.cid;
                mySS.sfid = dsaRsp.sfid;

                // Ready for full operation
                mySS.status = (int)Status.Running;

                // Confirm that everything is OK.
                dsaRsp.confirmationCode = (int)CCValues.OK;
                dsaRsp.Encode();

                argFrame.SetFrame(argFrame.cid, (int)mgmtType.DSA_RSP,
dsaRsp.byteArray);
```

```
                Console.WriteLine("[Wimax.DSA_RSPHandler] Sending DSA_RSP:
cid:{0}, transactionId:{1}\nconfirmationCode:{2}, Transport CID:{3}, sfid:{4}",
                    argFrame.cid, dsaRsp.transactionId, dsaRsp.confirmationCode,
dsaRsp.cid, dsaRsp.sfid);

            }
            else if (mode == (int)Wimax.netMode.Client)
            {
                // Get packet
                dsaRsp.byteArray = argFrame.payload;
                dsaRsp.Decode();

                Console.WriteLine("[Wimax.DSA_RSPHandler] Received DSA_RSP:
cid:{0}, transactionId:{1}\nconfirmationCode:{2}, Transport CID:{3}, sfid:{4}",
                    argFrame.cid, dsaRsp.transactionId, dsaRsp.confirmationCode,
dsaRsp.cid, dsaRsp.sfid);

                // Check confirmation code

                // Get assigned CID and sfid
                mySS.transportCid = dsaRsp.cid;
                mySS.sfid = dsaRsp.sfid;
                // Ready for full operation
                mySS.status = (int)Status.Running;
            }
            else if (mode == (int)Wimax.netMode.Relay)
            {
                // Get packet
                dsaRsp.byteArray = argFrame.payload;
                dsaRsp.Decode();

                Console.WriteLine("[Wimax.DSA_RSPHandler] Relaying DSA_RSP:
cid:{0}, transactionId:{1}\nconfirmationCode:{2}, Transport CID:{3}, sfid:{4}",
                    argFrame.cid, dsaRsp.transactionId, dsaRsp.confirmationCode,
dsaRsp.cid, dsaRsp.sfid);

                mySS.transportCid = dsaRsp.cid;
                mySS.sfid = dsaRsp.sfid;
                // Ready for full operation
                mySS.status = (int)Status.Running;
            }

            return (int)ReturnVal.OK;
        }

        // DSA_ACK handler interface for server
        public int DSA_ACKServer(MgmtFrame inFrame, EntryBS argBS)
        {
            DSA_ACK dsaAck = new DSA_ACK(entry);
            return DSA_ACKHandler(ref inFrame, ref dsaAck, argBS, null);
        }

        public int DSA_ACKRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            return (int)ReturnVal.OK;
        }

        // The DSA_ACK handler
        protected int DSA_ACKHandler(ref MgmtFrame argFrame, ref DSA_ACK dsaAck,
EntryBS argBS, EntryRS argRS)
        {
            if (!(entry is EntrySS))
            {
                throw new Exception("Wimax.DSA_ACKHandler: Entry must be
EntrySS");
            }
            EntrySS mySS = (EntrySS)entry;

            if (mode == (int)Wimax.netMode.Client)
            {
                dsaAck.transactionId = mySS.transactionID;
                dsaAck.confirmationCode = (int)CCValues.OK;
                dsaAck.Encode();

                Console.WriteLine("[Wimax.DSA_ACKHandler] Sent DSA_ACK: cid:{0},
transactionId:{1}, confirmationCode:{2}",
                    argFrame.cid, dsaAck.transactionId,
dsaAck.confirmationCode);

                argFrame.SetFrame(mySS.primaryCid, (int)mgmtType.DSA_ACK,
dsaAck.byteArray);
            }
            else if (mode == (int)Wimax.netMode.Server)
            {
                dsaAck.byteArray = argFrame.payload;
                dsaAck.Decode();

                Console.WriteLine("[Wimax.DSA_ACKHandler] Read DSA_ACK: cid:{0},
transactionId:{1}, confirmationCode:{2}",
                    argFrame.cid, dsaAck.transactionId,
dsaAck.confirmationCode);

            }
            return (int)ReturnVal.OK;
        }

        // DSX_RVD handler for the client and server
        public int DSX_RVDServer (MgmtFrame inFrame, out MgmtFrame outFrame,
EntryBS argBS)
        {
            outFrame = new MgmtFrame();
            // Respond to DSA_REQ
            // Send DSX_RVD
            // Send DSA_RSP
            return (int)ReturnVal.OK;
        }

        public int DSX_RVDClient(MgmtFrame inFrame)
        {
            DSX_RVD dsxRvd = new DSX_RVD(entry);
            DSX_RVDHandler(ref inFrame, ref dsxRvd, null, null);
            return (int)ReturnVal.OK;
        }

        public int DSX_RVDRelay(MgmtFrame inFrame, EntryRS argRS)
        {
            return (int)ReturnVal.OK;
        }

        // The DSX_RVD handler
        protected int DSX_RVDHandler(ref MgmtFrame argFrame, ref DSX_RVD dsxRvd,
EntryBS argBS, EntryRS argRS)
        {
            if (!(entry is EntrySS))
            {
                throw new Exception("Wimax.DSX_RVDHandler: Entry must be
EntrySS");
            }
            EntrySS mySS = (EntrySS)entry;

            if (mode == (int)Wimax.netMode.Server)
            {
                dsxRvd.transactionId = mySS.transactionID;
                dsxRvd.confirmationCode = (int)CCValues.OK;
                dsxRvd.Encode();

                Console.WriteLine("[Wimax.DSX_RVDHandler] Sending DSX_RVD:
cid:{0}, transactionId:{1}, confirmationCode:{2}",
                    argFrame.cid, dsxRvd.transactionId,
dsxRvd.confirmationCode);

                argFrame.SetFrame(argFrame.cid, (int)mgmtType.DSX_RVD,
dsxRvd.byteArray);

            }
            else if (mode == (int)Wimax.netMode.Client)
            {
                dsxRvd.byteArray = argFrame.payload;
                dsxRvd.Decode();

                Console.WriteLine("[Wimax.DSX_RVDHandler] Received DSX_RVD:
cid:{0}, transactionId:{1}, confirmationCode:{2}",
                    argFrame.cid, dsxRvd.transactionId,
dsxRvd.confirmationCode);
            }
            return (int)ReturnVal.OK;
        }
    }
}
```

B.6.4 WimaxRes.cs

```
/* Name: WimaxRes.cs
 * The implimentation of WiMAX for communication both between
 * the BS and RS nodes (Wimax over DSRC), as well as between
 * the RS nodes and the BS node (uplink using 16j tunneling).
 *
 * This file is responsible for the allocation of system
 * resources (CIDs and BSIDs), used by the BS node in a
 * given WiMAX cell */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public class WimaxRes
    {
        private bool[] cidList;
        private CidEntry[] cidEntries;

        // Values taken from IEEE 802.16j
        private const int CidStart = 0;
        private const int CidFinish = 65535;
        private const int CidFirst = 1;
        private const int CidM = 13036;
        private const int CidN = 39108;
        private const int CidP = 52144;
        private const int CidLast = 65138;
        private const int CidTypes = 5;

        private bool[] bsidList;


        // Using BSID of xxx00, while xxx01-xxx99 are the
        // BSIDs for RS serving under it
        private const int BsidStart = 0;
        private const int BsidEnd = 98;
        private const int BsidOffset = 1; // Actually 1-99
        private int bsid;

        // The categories of CIDs
        public enum CidType : int
        {
            Basic = 0,
            Primary = 1,
            Tunnel = 2,
            ManagedTunnel = 3,
            Transport = 4
        }

        // Entry in the CID allocation table
        class CidEntry
        {
            public int first;
            public int last;
        }
        // Set up the allocation system for CIDs within the
        // WiMAX cell (used by BS node). Also allocates
        // BSIDs for RS nodes.
        public WimaxRes(int argBsid)
        {
            CidEntry tempEntry;

            // Initialize the CID and BSID lists
            cidList = new bool[CidFinish + 1];
            bsidList = new bool[BsidEnd + 1];
            cidEntries = new CidEntry[CidTypes];

            bsid = argBsid;

            // Initialize each entry type
            // These formulas are from IEEE 802.16j

            tempEntry = new CidEntry();
            tempEntry.first = CidFirst;
            tempEntry.last = CidM;
            cidEntries[(int)CidType.Basic] = tempEntry;

            tempEntry = new CidEntry();
            tempEntry.first = CidM + 1;
            tempEntry.last = (2 * CidM);
            cidEntries[(int)CidType.Primary] = tempEntry;

            tempEntry = new CidEntry();
            tempEntry.first = (2 * CidM) + 1;
            tempEntry.last = CidN;
            cidEntries[(int)CidType.Tunnel] = tempEntry;

            tempEntry = new CidEntry();
            tempEntry.first = CidN + 1;
            tempEntry.last = CidP;
            cidEntries[(int)CidType.ManagedTunnel] = tempEntry;

            tempEntry = new CidEntry();
            tempEntry.first = CidP + 1;
            tempEntry.last = CidLast;
            cidEntries[(int)CidType.Transport] = tempEntry;
        }

        // Request a new CID for a user of a given type
        public int GetCid(int argType)
        {
            // default response if we fail to find a CID
            int retval = (int)ReturnVal.Fail;

            lock(cidList)
            {
                for (int c = cidEntries[argType].first; c <=
cidEntries[argType].last; c++)
                {
                    if (cidList[c] == false)
                    {
                        cidList[c] = true;
                        retval = c;
                        break;
                    }
                }
            }

            return retval;
        }

        // Return a CID no longer in use
        public int PutCid(int argCid)
        {
            int retval;
            lock (cidList)
            {
                if (cidList[argCid] == true)
                {
                    cidList[argCid] = false;
                    retval = (int)ReturnVal.OK;
                }
                else
                {
                    retval = (int)ReturnVal.Fail;
                }
            }

            return retval;
        }

        // Find a new BSID for an RS node
        public int getBsid()
        {
            // default response if we fail to find a BSID
            int retval = (int)ReturnVal.Fail;

            lock (bsidList)
            {
                for (int c = BsidStart; c <= BsidEnd; c++)
                {
                    if (bsidList[c] == false)
                    {
                        bsidList[c] = true;
                        retval = c + BsidOffset + bsid;
                        break;
                    }
                }
            }

            return retval;
        }

        // Return a BSID no longer in use
        public int putBsid(int argBsid)
```

```
        {
            int retval;

            lock (bsidList)
            {
                if (bsidList[argBsid] == true)
                {
                    bsidList[argBsid] = false;
                    retval = (int)ReturnVal.OK;
                }
                else
                {
                    retval = (int)ReturnVal.Fail;
                }
            }
            return retval;
        }
    }
}
```

B.7 L3NP Network Protocol

B.7.1 L3np.cs

```
/* Name: L3np.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This is the main file */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {

        public L3np(EntrySS argSS)
            : base(netMode.Client, netType.AC)
        {
            phase = (int)Phases.LinkDead;
            ssEntry = argSS;
            sessionId = hsSid;
            Lcp = new LCP(this);
            Ipcp = new IPCP(this);
        }
        public L3np(EntryIAG argIAG)
            : base(netMode.Server, netType.NS)
        {
            phase = (int)Phases.LinkDead;
            iagEntry = argIAG;
            sessionId = hsSid;

            //The L3np resources is only used by the server
            myRes = new L3npRes();
            Lcp = new LCP(this);
            Ipcp = new IPCP(this);
        }

        // Generate a random transaction identifier
        public int GenerateIdentifier()
        {
            Random random = new Random();
            // Return 16-bit number
            return random.Next(0, 255);
        }

        // Generate a new transaction identifier (can't repeat)
        public int GenerateIdentifier(int oldID)
        {
            int retval = oldID;
            Random random = new Random();
            // Return 16-bit number

            while (retval == oldID)
            {
                retval = random.Next(0, 255);
            }

            return retval;
        }
    }
}
```

B.7.2 L3np.definitions.cs

```
/* Name: L3np.definitions.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the main L3NP definitions, including
 * the L3NP frame structure, etc. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        // Variables
        public int phase;
        protected EntrySS ssEntry;
        protected EntryIAG iagEntry;

        public int sessionId;
        public int ipAddress;

        protected L3npRes myRes;

        public const int hsSid = 0;
        static int bufferLength = 1024;

        // Links to derived classes
        public LCP Lcp;
        public IPCP Ipcp;

        // The states of the L3NP link
        public enum Phases : int
        {
            LinkDead,
            LinkEstablish,
            Authenticate,
            NetworkLayer,
            Handover,
            Terminate
        }

        // The protocol types currently recognized
        public enum Protocols : int
        {
            IP = 0x0021,
            IPCP = 0x8021,
            LCP = 0xc021,
            // These authentication protocols are
            // not currently processed.
            PAP = 0xc023,
            CHAP = 0xc223,
            EAP = 0xc227,

            // Only used internally. Easily recognized.
            DEAD = 0xdead
        }

        // From the PPP standard
        public enum Codes : int
        {
            ConfigureRequest = 1,
            ConfigureAck = 2,
            ConfigureNak = 3,
            ConfigureReject = 4,
            TerminateRequest = 5,
            TerminateAck = 6,
            CodeReject = 7,
            ProtocolReject = 8,
            EchoRequest = 9,
            EchoReply = 10,
            DiscardRequest = 11
        }

        // Generic Frame structure.
        public abstract class Frame
        {
            public int headerFields;
            public int headerSize;
            protected const int baseFields = 2;
            public int size;

            public byte[] byteArray = null;

            public int sessionid;
            public int protocol;
            public byte[] payload = null;

            // These are overridden in each inheriting class
            // Versions with no options
            public abstract int Encode();
            public abstract int Decode();

            // Versions with options
            // Versions with data
               public int Encode(params OptionBase[] argOpts)
                {
                   // pack the arguments
                    EncodeOptions(argOpts);

                   // Call the normal encode
                   return Encode();
                }

                public int Decode(out OptionBase[] argOpts)
                {
                    // Call the orignal decode
                    int retval = Decode();

                    // unpack the arguments
                    DecodeOptions(out argOpts);
                    return retval;
                }

            // Takes the fields + payload and puts it into byteArray
            protected int EncodePacket(params int[] argVals)
            {
                int c = 0;
```

```
                if (payload == null)
                {
                    size = headerSize;
                }
                else
                {
                    size = headerSize + payload.Length;
                }

                byteArray = new byte[size];

                // Copy the common information for all protocols
                (System.BitConverter.GetBytes(sessionid)).CopyTo(byteArray, c);
                c += fieldSize;

                (System.BitConverter.GetBytes(protocol)).CopyTo(byteArray, c);
                c += fieldSize;

                // Copy the additional information for each protocol
                foreach (int i in argVals)
                {
                    (System.BitConverter.GetBytes(i)).CopyTo(byteArray, c);
                    c += fieldSize;
                }

                // Finally, copy in the payload
                if (payload != null)
                {
                    Array.Copy(payload, 0, byteArray, c, payload.Length);
                }

                return (int)ReturnVal.OK;
            }

            // Uses byteArray to populate the fields + payload
            protected int DecodePacket(out int[] argVals)
            {
                int c = 0;
                int payloadLength = byteArray.Length - headerSize;
                Console.WriteLine("[L3np.Frame.DecodePacket]:
ByteArray.Length:{0}, headerSize:{1}, payloadLength:{2}",
                    byteArray.Length, headerSize, payloadLength);

                size = byteArray.Length;

                argVals = new int[headerFields];

                // Copy fields common to all protocols
                sessionid = System.BitConverter.ToInt32(byteArray, c);
                c += fieldSize;

                protocol = System.BitConverter.ToInt32(byteArray, c);
                c += fieldSize;
                // Copy protocol specific fields
                if (headerFields > 0)
                {
                    for (int i = 0; i < headerFields; i++)
                    {
                        argVals[i] = System.BitConverter.ToInt32(byteArray, c);
                        c += fieldSize;
                    }
                }

                // Copy the remaining data to the payload
                if (payloadLength > 0)
                {
                    payload = new byte[payloadLength];
                    Array.Copy(byteArray, c, payload, 0, payloadLength);
                    Console.WriteLine("[L3np.Frame.DecodePacket]: Copied {0}
bytes payload", payloadLength);
                }

                return (int)ReturnVal.OK;
            }

            protected int EncodeOptions(OptionBase[] argOptions)
            {
                byte[] optionBuffer = new byte[bufferLength];
                int optionCount = 0;
                int optionIndex = 0;
                int c;

                payload = null;

                // Put the encoded data of each option into the buffer
                foreach (OptionBase o in argOptions)
                {
                    o.Encode();
                    optionCount++;
                    o.byteArray.CopyTo(optionBuffer, optionIndex);
                    optionIndex += o.byteArray.Length;
                }

                Console.WriteLine("[L3np.Frame.EncodeOptions] Encoded {0}
options, {1} bytes",
                    optionCount, optionIndex);

                // Create storage space in payload and copy over the buffer
                payload = new byte[optionIndex];

                for (c = 0; c < optionIndex; c++)
                {
                    payload[c] = optionBuffer[c];
                }

                return (int)ReturnVal.OK;
            }
            // Turn payload into options
            // Not currently implimented
            protected int DecodeOptions(out OptionBase[] argOptions)
            {
                return (int)ReturnVal.ERROR;
            }

        }

        // The PPPFrame is a generic frame style, used to decode
        // the header before the type is determined.
        public class PPPFrame : Frame
        {
            public PPPFrame()
            {
                headerFields = 0;
                headerSize = baseFields * fieldSize;
                protocol = (int)Protocols.DEAD;
            }

            // We should never be encoding these sort of messages
            public override int Encode()
            {
                //return EncodePacket(code, identifier, length);
                throw new Exception("[L3np.PPPFrame.Encode] This type should
never be encoded");
            }

            // Only decodes the common header
            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                return retVal;
            }
        }

        // The general definition of an Option, the data
        // format used in LCP and IPCP, as defined
        // by the PPP standard.
        public abstract class OptionBase
        {
            public int headerFields;
            public int headerSize;
            //Replaced with the length field
            //public int size;
            protected const int baseFields = 2;
            protected const int genType = 255;

            public byte[] byteArray;

            public int type;
            public int length;
```

```
// Payload will be mostly unused, instead using parameters
public byte[] payload = null;

// These are overridden in each inheriting class
// Versions with no data
public abstract int Encode();
public abstract int Decode();

// Takes the fields + payload and puts it into byteArray
public int EncodePacket(params int[] argVals)
{
    byteArray = new byte[length];
    int c = 0;

    if (payload == null)
    {
        length = headerSize;
    }
    else
    {
        length = headerSize + payload.Length;
    }

    byteArray = new byte[length];

    // Copy the common information for all protocols
    (System.BitConverter.GetBytes(type)).CopyTo(byteArray, c);
    c += fieldSize;

    (System.BitConverter.GetBytes(length)).CopyTo(byteArray, c);
    c += fieldSize;

    // Copy the additional information for each protocol
    foreach (int i in argVals)
    {
        (System.BitConverter.GetBytes(i)).CopyTo(byteArray, c);
        c += fieldSize;
    }

    // Finally, copy in the payload (if defined)
    if (payload != null)
    {
        Array.Copy(payload, 0, byteArray, c, payload.Length);
    }

    return (int)ReturnVal.OK;
}

// Uses byteArray to populate the fields + payload
public int DecodePacket(out int[] argVals)
{
    int c = 0;
    int payloadLength = byteArray.Length - headerSize;

    argVals = new int[headerFields];
```

```
                // Copy fields common to all protocols
                type = System.BitConverter.ToInt32(byteArray, c);
                c += fieldSize;

                length = System.BitConverter.ToInt32(byteArray, c);
                c += fieldSize;

                // Copy protocol specific fields
                for(int i=0; i < headerFields; i++)
                {
                    argVals[i] = System.BitConverter.ToInt32(byteArray, c);
                    c += fieldSize;
                }

                // Copy the remaining data to the payload
                payload = new byte[payloadLength];
                Array.Copy(byteArray, c, payload, 0, payloadLength);

                return (int)ReturnVal.OK;
            }
        }

        // The generic option header.
        // Used to read the common header fields to determine type
        public class GenericOption : OptionBase
        {

            public GenericOption()
            {
                headerFields = 0;
                headerSize = baseFields * fieldSize;
                type = genType;
            }

            public override int Encode()
            {
                throw new Exception("[L3np.GenericOption.Encode] This type
should never be encoded");
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                return retVal;
            }
        }
    }
}
```

B.7.3 L3np.handlers.cs

```
/* Name: L3np.handlers.cs
```

```
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the generic L3NP handlers, of which
 * there is currently none. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
    }
}
```

B.7.4 L3npRes.cs

```
/* Name: L3npRes.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the resources requires for the NS.
 * This includes managing the IP addresses and the
 * Session IDs (sid) assigned to clients */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public class L3npRes
    {
        private bool[] ipList;
        private bool[] sidList;

        private const int ipFirst = 1;
        private const int ipLast = 250;
        private const string ipPrefix = "192.168.100.";
        private readonly int ipNumber;
        private readonly int ipOffset;

        private const int sidFirst = 1;
        private const int sidLast = 255;
        private readonly int sidNumber;
        private readonly int sidOffset;


        public L3npRes()
        {
            ipNumber = ipLast - ipFirst;
            ipOffset = ipFirst;
            sidNumber = sidLast - sidFirst;
            sidOffset = ipFirst;

            ipList = new bool[ipNumber];
            sidList = new bool[sidNumber];

        }


        // Assign the next avaliable IP address
        public int GetIp()
        {
            // The default response if we fail to find one
            int retval = (int)ReturnVal.Fail;
            System.Net.IPAddress newIP;
            byte[] ipArray;
            int ipEnd;

            lock (ipList)
            {
                // Find the first avaliable entry
                for (int c = 0; c < ipNumber; c++)
                {
                    if (ipList[c] == false)
                    {
                        ipList[c] = true;

                        // Apply the offet to get the actual IP
                        ipEnd = c + ipOffset;
                        newIP = System.Net.IPAddress.Parse(ipPrefix +
ipEnd.ToString());
                        ipArray = newIP.GetAddressBytes();
                        retval = System.BitConverter.ToInt32(ipArray, 0);

                        Console.WriteLine("[L3npRes.GetIP] Allocated new IP
address: {0}", newIP.ToString());
                        break;
                    }
                }
            }

            return retval;
        }

        // Free an IP address no longer in use.
        public int PutIP(int argIP)
        {
            int retval;

            // Check to see if the IP address is valid
            if ((argIP < ipFirst) || (argIP > ipLast))
            {
                return (int)ReturnVal.Error;
            }

            lock (ipList)
            {
                // Check to see if that IP address is assigned
                if (ipList[argIP - ipOffset] == true)
                {
                    // If so, unassign and return OK
                    ipList[argIP - ipOffset] = false;
                    retval = (int)ReturnVal.OK;
                }
                else
                {
                    retval = (int)ReturnVal.Fail;
                }
            }
            return retval;
        }

        // Assign the next avaliable Session ID
        public int GetSid()
        {
            // The default response if we fail to find one
            int retval = (int)ReturnVal.Fail;

            lock (sidList)
            {
                // Find the first avaliable entry
                for (int c = 0; c < sidNumber; c++)
                {
                    if (sidList[c] == false)
                    {
                        sidList[c] = true;

                        // Apply the offet to get the actual session ID
                        retval = c + sidOffset;
                        break;
                    }
                }
            }

            return retval;
        }
```

```
        // Free a session ID no longer in use.
        public int PutSid(int argSid)
        {
            int retval;

            // Check to see if the IP address is valid
            if ((argSid < sidFirst) || (argSid > sidLast))
            {
                return (int)ReturnVal.Error;
            }

            lock (ipList)
            {
                // Check to see if that IP address is assigned
                if (ipList[argSid - sidOffset] == true)
                {
                    // If so, unassign and return OK
                    ipList[argSid - sidOffset] = false;
                    retval = (int)ReturnVal.OK;
                }
                else
                {
                    retval = (int)ReturnVal.Fail;
                }
            }
            return retval;
        }
    }
}
```

B.7.5 LCP.cs

```
/* Name: LCP.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the main module for the Link Control
 * Protocol (LCP) functionality within the L3NP protocol,
 * based on the PPP standard. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        public partial class LCP
```

```
{
    private L3np myL3np;

    // Constructor
    public LCP(L3np argL3np)
    {
        myL3np = argL3np;
    }

    // Define an LCP frame structure
    public class LCPFrame : Frame
    {
        public int code;
        public int identifier;
        public int length;

        public LCPFrame()
        {
            headerFields = 3;
            headerSize = (headerFields + baseFields) * fieldSize;
            protocol = (int)Protocols.LCP;
        }

        public LCPFrame(int argCode) : this()
        {
            code = argCode;
        }

        public override int Encode()
        {
            length = headerFields * fieldSize;
            return EncodePacket(code, identifier, length);
        }

        public override int Decode()
        {
            int[] fields;

            int retVal = DecodePacket(out fields);

            code = fields[0];
            identifier = fields[1];
            length = fields[2];

            return retVal;
        }
    }

    // The options types supported by LCP
    // (none currently implimented)
    public enum OptionType : int
    {
        Reserved = 0,
        MaximumReceiveUnit = 1,
        AuthenticationProtocol = 3,
                QualityProtocol = 4,
                MagicNumber = 5,
                ProtocolFieldCompression = 7,
                AddressAndControlFieldCompression = 8
            }
        }
    }
}
```

B.7.6 LCP.handlers.cs

```
/* Name: LCP.handlers.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the message handlers for the LCP
 * functionality within L3NP */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        public partial class LCP
        {
            // Client and server interfaces to the ConfigureRequest handler.
            public int ConfigureRequestClient(out LCPFrame outFrame)
            {
                outFrame = new LCPFrame((int)Codes.ConfigureRequest);
                return ConfigureRequestHandler(ref outFrame);
            }

            public int ConfigureRequestServer(LCPFrame inFrame, out LCPFrame
outFrame, out int outSid)
            {
                int retval;
                outFrame = new LCPFrame((int)Codes.ConfigureAck);

                ConfigureRequestHandler(ref inFrame);

                outFrame.identifier = inFrame.identifier;

                retval = ConfigureAckHandler(ref outFrame);
                outSid = myL3np.sessionId;
                return retval;
            }
            // The ConfigureRequest handler
            protected int ConfigureRequestHandler(ref LCPFrame argFrame)
            {
                if (myL3np.mode == (int)L3np.netMode.Client)
                {
                    argFrame.identifier = myL3np.GenerateIdentifier();
                    argFrame.sessionid = myL3np.sessionId;
                    argFrame.Encode();
                    Console.WriteLine("[LCP.ConfigureRequestHandler] Sending
Request");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);

                }
                else if (myL3np.mode == (int)L3np.netMode.Server)
                {
                    argFrame.Decode();
                    Console.WriteLine("[LCP.ConfigureRequestHandler] Request
Received");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                return (int)ReturnVal.OK;
            }

            // Client interface for the ConfigureAck handler
            public int ConfigureAckClient(LCPFrame inFrame)
            {
                return ConfigureAckHandler(ref inFrame);
            }

            // Doesn't do anything.
            public int ConfigureAckServer()
            {
                return (int)ReturnVal.Error;
            }

            // The ConfigureAck handler
            protected int ConfigureAckHandler(ref LCPFrame argFrame)
            {
                if (myL3np.mode == (int)L3np.netMode.Client)
                {
                    argFrame.Decode();

                    // Store new session id
                    myL3np.sessionId = argFrame.sessionid;
                    Console.WriteLine("[LCP.ConfigureAckHandler] Request
Received");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                else if (myL3np.mode == (int)L3np.netMode.Server)
                {
                    // Create session id
                    myL3np.sessionId = myL3np.myRes.GetSid();
                    argFrame.sessionid = myL3np.sessionId;
                    Console.WriteLine("[LCP.ConfigureAckHandler] Assigned new
session ID: {0}", myL3np.sessionId);
                    // Encode message
                    argFrame.Encode();
                    Console.WriteLine("[LCP.ConfigureAckHandler] Sending
Request");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                return (int)ReturnVal.OK;
            }
        }
    }
}
```

B.7.7 IPCP.cs

```
/* Name: IPCP.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the main module for the Internet
 * Protocol Control Protocol (IPCP) functionality within the
 * L3NP protocol, based on the PPP standard. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        public partial class IPCP
        {
            private L3np myL3np;

            // Constructor
            public IPCP(L3np argL3np)
            {
                myL3np = argL3np;
            }

            // The IPCP frame structure
            public class IPCPFrame : Frame
            {
                public int code;
                public int identifier;
                public int length;
                public IPCPFrame()
                {
                    headerFields = 3;
                    headerSize = (headerFields + baseFields) * fieldSize;
                    protocol = (int)Protocols.IPCP;
                }

                public IPCPFrame(int argCode) : this()
                {
                    code = argCode;
                }


                public override int Encode()
                {
                    length = headerFields * fieldSize;
                    return EncodePacket(code, identifier, length);
                }

                public override int Decode()
                {
                    int[] fields;

                    int retVal = DecodePacket(out fields);

                    code = fields[0];
                    identifier = fields[1];
                    length = fields[2];

                    return retVal;
                }
           }

            // Option types supported by IPCP
            public enum OptionType : int
            {
                IpAddresses = 1,
                IPCompressionProtocol = 2,
                // The one one currently supported
                IPAddress = 3
            }

            // The structure for an IP Address option
            public class IPAddress : OptionBase
            {
                public int ipAddress;

                public IPAddress()
                {
                    headerFields = 1;
                    headerSize = (headerFields + baseFields) * fieldSize;
                    type = (int)OptionType.IpAddresses;
                }

                public override int Encode()
```

```
                {
                    return EncodePacket(ipAddress);
                }

                public override int Decode()
                {
                    int[] fields;

                    int retVal = DecodePacket(out fields);

                    ipAddress = fields[0];

                    return retVal;
                }
            }
        }
    }
}
```

B.7.8 IPCP.handlers.cs

```
/* Name: IPCP.handlers.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the message handlers for the IPCP
 * functionality within L3NP */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        public partial class IPCP
        {
            // Client and server interfaces to the ConfigureRequest handler
            public int ConfigureRequestClient(out IPCPFrame outFrame)
            {
                outFrame = new IPCPFrame((int)Codes.ConfigureRequest);
                return ConfigureRequestHandler(ref outFrame);

            }

            public int ConfigureRequestServer(IPCPFrame inFrame, out IPCPFrame
outFrame)
            {
                outFrame = new IPCPFrame((int)Codes.ConfigureAck);

                ConfigureRequestHandler(ref inFrame);

                outFrame.identifier = inFrame.identifier;
                outFrame.sessionid = inFrame.sessionid;
                return ConfigureAckHandler(ref outFrame);
            }

            // ConfigureRequest handler
            protected int ConfigureRequestHandler(ref IPCPFrame argFrame)
            {
                if (myL3np.mode == (int)L3np.netMode.Client)
                {
                    argFrame.identifier = myL3np.GenerateIdentifier();
                    argFrame.sessionid = myL3np.sessionId;
                    argFrame.Encode();
                    Console.WriteLine("[IPCP.ConfigureRequestHandler] Sending
Request");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);

                }
                else if (myL3np.mode == (int)L3np.netMode.Server)
                {
                    argFrame.Decode();
                    Console.WriteLine("[IPCP.ConfigureRequestHandler] Request
Received");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                return (int)ReturnVal.OK;
            }

            // Client interface into the ConfigureAck handler
            public int ConfigureAckClient(IPCPFrame inFrame)
            {
                return ConfigureAckHandler(ref inFrame);
            }

            public int ConfigureAckServer()
            {
                return (int)ReturnVal.Error;
            }

            // ConfigureAck handler
            protected int ConfigureAckHandler(ref IPCPFrame argFrame)
            {
                if (myL3np.mode == (int)L3np.netMode.Client)
                {
                    argFrame.Decode();
                    Console.WriteLine("[IPCP.ConfigureAckHandler] Request
Received");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                else if (myL3np.mode == (int)L3np.netMode.Server)
                {
                    // Create session id
                    IPAddress optIP = new IPAddress();

                    myL3np.ipAddress = myL3np.myRes.GetIp();
                    optIP.ipAddress = myL3np.ipAddress;

                    argFrame.sessionid = myL3np.sessionId;
                    Console.WriteLine("[LCP.ConfigureAckHandler] Assigned new
IP: {0}", myL3np.ipAddress);

                    argFrame.Encode(optIP);
                    Console.WriteLine("[IPCP.ConfigureAckHandler] Sending
Request");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                return (int)ReturnVal.OK;
            }

            // Client interface to the ConfigureNak handler
            public int ConfigureNakClient(IPCPFrame inFrame)
            {
                return ConfigureNakHandler(ref inFrame);
            }

            public int ConfigureNakServer()
            {
                return (int)ReturnVal.Error;
            }

            // ConfigureNak Handler
            protected int ConfigureNakHandler(ref IPCPFrame argFrame)
            {
                if (myL3np.mode == (int)L3np.netMode.Client)
                {
                    argFrame.Decode();
                    Console.WriteLine("[IPCP.ConfigureNakHandler] Request
Received");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                else if (myL3np.mode == (int)L3np.netMode.Server)
                {
                    // Create session id
                    int newIP;
                    myL3np.ipAddress = myL3np.myRes.GetIp();
                    newIP = myL3np.ipAddress;

                    argFrame.sessionid = myL3np.sessionId;
                    Console.WriteLine("[LCP.ConfigureNakHandler] Assigned new
IP: {0}", myL3np.ipAddress);

                    argFrame.Encode();
                    Console.WriteLine("[IPCP.ConfigureNakHandler] Sending
Request");
                    Console.WriteLine("Identifier: {0}", argFrame.identifier);
                }
                return (int)ReturnVal.OK;
            }
        }
    }
}
```

B.7.9 IP.cs

```
/* Name: IP.cs
 * The implimentation of L3NP for communication between
 * the AC and NS nodes. This network layer protocol operates
 * over top of the underlying WiMAX/WBNP links. As such,
 * it doesn't establish any actual TCP/IP links, but uses
 * those in the underlying network layers.
 *
 * The structure of the L3NP module breaks apart handlers
 * for LCP, IPCP and IP frames into seperate sub-modules.
 *
 * This file contains the support for IP traffic within L3NP */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class L3np : NetBase
    {
        public partial class IP
        {
            // The definition of an IP traffic frame
            public class IPFrame : Frame
            {
                public IPFrame()
                {
                    headerFields = 0;
                    headerSize = baseFields * fieldSize;
                    protocol = (int)Protocols.IP;
                }

                public override int Encode()
                {
                    return EncodePacket();
                }

                public override int Decode()
                {
                    int[] fields;

                    int retVal = DecodePacket(out fields);

                    return retVal;
                }
            }
        }
    }
}
```

B.8 WBNP Network Protocol

B.8.1 NetBack.cs

```
/* Name: NetBack.cs
 * The implimentation of the WiMAX Backend Network Protocol
 * for communications between the BS nodes and the IAG
 * node across the WiMAX service provider's backend
 * network.
 *
 * This is the main file */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public partial class NetBack : NetBase
    {
        // Class variables
        protected int bsid;
        protected EntryBS entry;

        // Constructors
        public NetBack(netMode argMode, netType argType, EntryBS argEntry) :
this(argMode, argType, argEntry, null) {}

        public NetBack(netMode argMode, netType argType, EntryBS argEntry,
Socket argSocket) :
            base(argMode, argType, argSocket)
        {
            entry = argEntry;
        }

        public int Send(Frame argFrame)
        {
            Data sendData;
            SendEncode(argFrame, out sendData);

            return Send(sendData);
        }

        public int AsyncSend(Frame argFrame, EntryBase argEntry)
        {
            Data sendData;
            SendEncode(argFrame, out sendData);

            lock(argEntry.sendQueue)
            {
                argEntry.sendQueue.Enqueue(sendData);
            }

            return (int)ReturnVal.OK;
        }


        public int SendEncode(Frame argFrame, out Data argData)
        {
            int c = 0;
            byte[] sendBuffer = new byte[argFrame.length];
            argFrame.length = (3 * fieldSize) + argFrame.payload.Length;

            System.BitConverter.GetBytes(argFrame.type).CopyTo(sendBuffer, 0);
            c += fieldSize;

            System.BitConverter.GetBytes(argFrame.length).CopyTo(sendBuffer, c);
            c += fieldSize;

            System.BitConverter.GetBytes(argFrame.cid).CopyTo(sendBuffer, c);
            c += fieldSize;

            argFrame.payload.CopyTo(sendBuffer, c);
            argData = new NetBase.Data(sendBuffer);

            return (int)ReturnVal.OK;
        }

        public int Read(Frame argFrame)
        {
            int returnVal;

            Data readData = new Data();
            returnVal = Read(readData);

            ReadDecode(argFrame, readData);
            return returnVal;
        }

        public int AsyncRead(Data argData, EntryBase argEntry, IAsyncResult ar,
Frame argFrame)
        {
            AsyncRead(argData, argEntry, ar);
            ReadDecode(argFrame, argData);

            return (int)ReturnVal.OK;
        }

        public int ReadDecode(Frame argFrame, Data argData)
        {
            int c = 0;
            int payloadLength;
            byte[] tempArray = new byte[fieldSize];

            Array.Copy(argData.payload, 0, tempArray, 0, fieldSize);
            c += fieldSize;
            argFrame.type = System.BitConverter.ToInt32(tempArray, 0);

            Array.Copy(argData.payload, c, tempArray, 0, fieldSize);
            c += fieldSize;
            argFrame.length = System.BitConverter.ToInt32(tempArray, 0);

            Array.Copy(argData.payload, c, tempArray, 0, fieldSize);
            c += fieldSize;
            argFrame.cid = System.BitConverter.ToInt32(tempArray, 0);

            payloadLength = argFrame.length - (3 * fieldSize);

            argFrame.payload = new byte[payloadLength];
            Array.Copy(argData.payload, c, argFrame.payload, 0, payloadLength);

            return (int) ReturnVal.OK;
        }
    }
}
```

B.8.2 NetBack.definitions.cs

```
/* Name: NetBack.definitions.cs
 * The implimentation of the WiMAX Backend Network Protocol
 * for communications between the BS nodes and the IAG
 * node across the WiMAX service provider's backend
 * network.
 *
 * This file contains definitions of resources, primarily
 * frame and message structures. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class NetBack : NetBase
    {
        // The WBNP frame definition
        public class Frame
        {
            public int type;
            public int length;
            public int cid;

            public byte[] payload;

            public Frame() { }
            public Frame(int argType, int argCid, byte[] argPayload)
            {
                type = argType;
                cid = argCid;
                payload = argPayload;
                length = (3 * fieldSize) + payload.Length;
            }

            public int SetFrame(int argType, int argCid, byte[] argPayload)
            {
                type = argType;
                cid = argCid;
                payload = argPayload;

                length = (3 * fieldSize) + payload.Length;
                return (int)ReturnVal.OK;
            }

        }

        protected const int noCid = 0;

        // The message types, as defined by the specification
        // in the thesis.
        public enum MessageType
        {
            // DATA is user IP traffic
            DATA                = 1,
            HS_REQ              = 2,
            HS_RSP              = 3,
            // The following are not yet implimented
            TERM_REQ            = 4,
            ERR_NOROUTE         = 5
        }

        // The message structure for non-DATA types
        abstract public class Message
        {
            public byte[] byteArray;
            protected int fields;
            protected int size;
            protected EntryBase entry;

            abstract public int Encode();
            abstract public int Decode();

            public int EncodePacket(params int[] argVals)
            {
                byteArray = new byte[size];
                int c = 0;

                foreach (int i in argVals)
                {
                    (System.BitConverter.GetBytes(i)).CopyTo(byteArray, c);
                    c += fieldSize;
                }

                return (int)ReturnVal.OK;
            }
            public int DecodePacket(out int[] argVals)
            {
                int c, d = 0;
                argVals = new int[fields];

                for (c = 0; c < fields; c++)
                {
                    argVals[c] = System.BitConverter.ToInt32(byteArray, d);
                    d += fieldSize;
                }

                return (int)ReturnVal.OK;
            }
        }

        // Message structure for HS_REQ messages
        public class HS_REQ : Message
        {
            public int version;
            public int bsid;

            public HS_REQ()
            {
                fields = 2;
                size = fields * fieldSize;
            }

            public override int Encode()
            {
                return EncodePacket(version, bsid);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                version = fields[0];
                bsid = fields[1];

                return retVal;
            }
        }

        // Message structure for HS_RSP messages
        public class HS_RSP : Message
        {
            public int version;

            public HS_RSP()
            {
                fields = 2;
                size = fields * fieldSize;
```

```
            }

            public override int Encode()
            {
                return EncodePacket(version);
            }

            public override int Decode()
            {
                int[] fields;

                int retVal = DecodePacket(out fields);

                version = fields[0];

                return retVal;
            }
        }
    }
}
```

B.8.3 NetBack.handlers.cs

```
/* Name: NetBack.handlers.cs
 * The implimentation of the WiMAX Backend Network Protocol
 * for communications between the BS nodes and the IAG
 * node across the WiMAX service provider's backend
 * network.
 *
 * This file stores event handlers for the WBNP communications.
 * These are called by the message handlers in the AC/NS modules. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    public partial class NetBack : NetBase
    {
        // Server and client interface to the HS_REQ handler
        public int HS_REQServer(Frame inFrame, out Frame outFrame, ref EntryIAG
argIAG)
        {
            outFrame = new Frame();
            HS_REQ hsReq = new HS_REQ();
            HS_RSP hsRsp = new HS_RSP();
            HS_REQHandler(ref inFrame, ref hsReq, argIAG);
            outFrame.cid = inFrame.cid;
            return HS_RSPHandler(ref outFrame, ref hsRsp, argIAG);
        }

        public int HS_REQClient(out Frame outFrame)
        {
            outFrame = new Frame();
            HS_REQ hsReq = new HS_REQ();

            return HS_REQHandler(ref outFrame, ref hsReq, null);
        }

        // HS_REQ handler
        protected int HS_REQHandler(ref Frame argFrame, ref HS_REQ hsReq,
EntryIAG argIAG)
        {
            if (mode == (int)netMode.Server)
            {
                hsReq.byteArray = argFrame.payload;
                hsReq.Decode();

                Console.WriteLine("[NetBack.HS_REQHandler] Received HS_REQ
frame, type:{0}, length:{1}, cid:{2}",
                    argFrame.type, argFrame.length, argFrame.cid);

                Console.WriteLine("bsid:{0}, version:{1}", hsReq.bsid,
hsReq.version);

                // Make local copies
                entry.bsid = hsReq.bsid;
                entry.version = hsReq.version;

            }
            else if (mode == (int)netMode.Client)
            {
                hsReq.bsid = entry.bsid;
                hsReq.version = Program.programVersion;
                hsReq.Encode();

                argFrame.SetFrame((int)MessageType.HS_REQ, noCid,
hsReq.byteArray);

                Console.WriteLine("[NetBack.HS_REQHandler] Sent HS_REQ frame,
type:{0}, length:{1}, cid:{2}",
                    argFrame.type, argFrame.length, argFrame.cid);

                Console.WriteLine("bsid:{0}, version:{1}", hsReq.bsid,
hsReq.version);
            }


            return (int)ReturnVal.OK;
        }

        // Client interface to the HS_RSP handler
        public int HS_RSPClient(Frame argFrame)
        {
            HS_RSP hsRsp = new HS_RSP();
            return HS_RSPHandler(ref argFrame, ref hsRsp, null);
        }

        // HS_RSP handler
        protected int HS_RSPHandler(ref Frame argFrame, ref HS_RSP hsRsp,
EntryIAG argIAG)
        {

            if (mode == (int)netMode.Client)
            {
                hsRsp.byteArray = argFrame.payload;
                hsRsp.Decode();

                Console.WriteLine("[NetBack.HS_RSPHandler] Received HS_RSP
frame, type:{0}, length:{1}, cid:{2}",
                    argFrame.type, argFrame.length, argFrame.cid);

                Console.WriteLine("version:{0}", hsRsp.version);
            }
            else if (mode == (int)netMode.Server)
            {
                hsRsp.version = Program.programVersion;
                hsRsp.Encode();

                argFrame.SetFrame((int)MessageType.HS_RSP, argFrame.cid,
hsRsp.byteArray);

                Console.WriteLine("[NetBack.HS_RSPHandler] Sent HS_RSP frame,
type:{0}, length:{1}, cid:{2}",
                    argFrame.type, argFrame.length, argFrame.cid);

                Console.WriteLine("version:{0}", hsRsp.version);
            }

            return (int)ReturnVal.OK;
        }

        // Server handler for DATA packets received at the NS node. At
        // this time, the demonstration system echos them back to the
        // source node.
        public int DATAServer(Frame inFrame, out Frame outFrame)
        {
            Console.WriteLine("[NetBack.DATAServer] Received DATA frame.
cid:{0}, type:{1}, length:{2} contents:\n->{3}<-",
                inFrame.cid, inFrame.type, inFrame.length,
stringEncoder.GetString(inFrame.payload));

            Console.WriteLine("Echoing message back...");
            outFrame = new Frame((int)MessageType.DATA, inFrame.cid,
inFrame.payload);

            return (int)ReturnVal.OK;

        }
    }
}
```

B.9 Module Entries

B.9.1 EntryBase.cs

```
/* Name: EntryBase.cs
 * The base class for other system node entry classes in the program. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WiMAXDSRC
{
    // The possible statuses for each node
    public enum Status : int
    {
        Unknown,
        Exit,
        Disconnected,
        Connected,
        Synced,
        Ranged,
        Authenticated,
        Registered,
        Running,
        Handover
    }

    // The abstract entry structure
    public abstract class EntryBase
    {
        public int status;
        public int rangingCode;
        public int version;

        // Send Queue
        public Queue<NetBase.Data> sendQueue = new Queue<NetBase.Data>();
        public byte[] readBuf;
    }
}
```

B.9.2 EntrySS.cs

```
/* Name: EntrySS.cs
 * The node entry for Subscriber Station (SS) nodes.
 * Stores various status information. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public class EntrySS : EntryBase
    {
        public string RSIp;
        public int RSPort;

        public Wimax wimax;
        public Socket socket;

        public int servingBSID;
        public EntryRS servingRS;

        public int basicCid;
        public int primaryCid;

        // OS mode
        public int osMode;

        // Variables for the service flow to contact
        // the IAG
        public int sfid;
        public int transportCid;
        public int transactionID;

        // Interconnect queue for the AC
        // This needs to be initialized before use.
        protected Queue<byte[]> appQueue = new Queue<byte[]>();

        // The property to allow AC access to the message queue.
        public byte[] AppQueue
        {
            get
            {
                byte[] tempData;
                lock (appQueue)
                {
                    tempData = appQueue.Dequeue();

                }
                return tempData;
            }

            set
            {
                lock (appQueue)
                {
                    appQueue.Enqueue(value);
                    Console.WriteLine("[EntrySS.AppQueue.set] Added to queue.
Total: {0}", appQueue.Count);
                }
            }
        }

        // Count the number of items waiting on the queue from AC
        public int AppQueueCount
        {
            get
            {
                return appQueue.Count;
            }
        }

    }
}
```

B.9.3 EntryRS.cs

```
/* Name: EntryRS.cs
 * The node entry for Relay Station (RS) nodes.
 * Stores various status information. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public class EntryRS : EntryBase
    {
        public string BSIp;
        public int BSPort;
        public int RSPort;

        public Socket bsSocket;
        public Wimax bsWimax;

        public Socket listenSocket;
        public Wimax listenWimax;

        // SS users serviced by this RS node.
        // Also used as a routing table.
        public LinkedList<EntrySS> ssConnections = new LinkedList<EntrySS>();

        // Specialized queue to handle traffic to relay
        public Queue<NetBase.Data> relayQueue = new Queue<NetBase.Data>();

        public int myBSID;
        public int servingBSID;

        public int basicCid;
        public int primaryCid;

        public int tCid;
        public int mtCid;

    }
}
```

B.9.4 EntryBS.cs

```
/* Name: EntryBS.cs
 * The node entry for Base Station (BS) nodes.
 * Stores various status information. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    public class EntryBS : EntryBase
    {
        public string iagIp;
        public int iagPort;

        public Socket iagSocket;
        public NetBack iagNetback;

        public Socket listenSocket;
        public Wimax listenWimax;

        // Lists of active RS and SS nodes in the system
        // Also used for routing purposes.
        public LinkedList<EntryRS> rsConnections = new LinkedList<EntryRS>();
        public LinkedList<EntrySS> ssConnections = new LinkedList<EntrySS>();

        public WimaxRes myRes;

        public int bsPort;
        public int bsid;
    }
}
```

B.9.5 EntryIAG.cs

```
/* Name: EntryBS.cs
 * The node entry for Internet Access
 * Gateway (IAG) nodes.
 * Stores various status information. */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Sockets;

namespace WiMAXDSRC
{
    // This entry does NOT derive from EntryBase,
    // since it doesn't use WiMAX.
    public class EntryIAG
    {
        public int iagPort;
        public NetBack netback;

        // The table of connected BS nodes.
        // Used for routing purposes.
        public LinkedList<EntryBS> bsConnections = new LinkedList<EntryBS>();

        // Interconnect queue for the NS module
        // This needs to be initialized before use.
        protected Queue<ModNS.InteropFrame> appQueue = new
Queue<ModNS.InteropFrame>();

        // The property to allow NS access to the message queue.
        public ModNS.InteropFrame AppQueue
        {
            get
            {
                ModNS.InteropFrame tempData;
                lock (appQueue)
                {
                    tempData = appQueue.Dequeue();

                }
                return tempData;
            }

            set
            {
                lock (appQueue)
                {
                    appQueue.Enqueue(value);
                    Console.WriteLine("[EntryIAG.AppQueue.set] Added to queue.
Total: {0}", appQueue.Count);
                }
            }
        }

        // Count the number of items waiting on the queue from NS
        public int AppQueueCount
        {
            get
            {
                return appQueue.Count;
            }
        }

        // Find a specific BS user by BSID
        public void FindBS(int argBsid, out EntryBS outBS)
        {
            Console.WriteLine("[L3np.FindBS] Looking for bsid: {0}", argBsid);
            lock (bsConnections)
            {
                outBS = bsConnections.First(delegate(EntryBS e)
                {
                    Console.WriteLine("[L3np.FindBS] Checking entry...");
                    if (e.bsid == argBsid)
                    {
                        Console.WriteLine("Entry Matches!");
                        return true;
                    }
                    else
                    {
                        Console.WriteLine("Entry does not match.");
                        return false;
                    }
                });
            }
        }
    }
}
```

References

[1] M. Sichitiu and M. Kihl, "Inter-Vehicle Communications Systems: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 2, pp. 88-105, Second Quarter 2008.

[2] Z. Tao, K. Teo, and J. Zhang, "Aggregation and Concatination in IEEE 802.16j Mobile Multihop Relay (MMR) Networks," in *IEEE Mobile WiMAX Symposium*, 2007, pp. 85-90.

[3] D. Jiang, V. Taliwal, A. Meier, W. Holfelder, and R. Herrtwich, "Design of 5.9 GHz DSRC-based Vehicular Safety Communications," *IEEE Wireless Communications Magazine*, pp. 36-43, Oct. 2006.

[4] WiMAX Forum, "Mobile WiMAX – Part I: A Technical Overview and Performance Evaluation," Aug. 2007.

[5] WiMAX Forum, "WiMAX Forum Mobile System Profile, Release 1.0 Approved Specification (Revision 1.4.0)," May 2007.

[6] IEEE, "Air Interface for Fixed Broadband Wireless Access Systems," *IEEE Standard 802.16e-2005 and 802.16-2004-Cor1-2005*, Feb. 2006.

[7] IEEE, "Air Interface for Fixed Broadband Wireless Access Systems – Multihop Relay Specification," *IEEE Standard 802.16j Draft 2*, Dec. 2007.

[8] P. Stevens and R. Heath, "The Future of WiMAX: Multihop Relaying with IEEE 802.16j," *IEEE Communications Magazine*, pp. 104-111, Jan. 2009.

[9] K. Fujimura and T. Hasegawa, "A Collaborative MAC Protocol for Inter-Vehicle and Road to Vehicle Communications," in *2004 IEEE Intelligent Transportation Systems Conference*, 2004, pp. 816-821.

[10] C. Perkins, "IP Mobility Support for IPv4," *IETF Standard RFC 3344*, Aug. 2002.

[11] A. Campbell et al., "Comparison of IP Micromobility Protocols," *IEEE Wireless Communications Magazine*, pp. 72-82, Feb. 2002.

[12] A. Valkó, "Cellular IP: A New Approach to Internet Host Mobility," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 1, pp. 50-65, Jan. 1999.

[13] W. Simpson, "The Point-to-Point Protocol (PPP)," *IETF Standard RFC 1661*, Jul. 1994.

[14] C. Rigney, S. Willens, A. Rubens, and W. Simpson, "Remote Authentication Dial In User Service," *IETF Standard RFC 2865*, Jun. 2000.

[15] L. Mamakos et al., "A Method for Transmitting PPP over Ethernet," *IETF Standard RFC 2516*, Feb. 1999.

[16] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," *IETF Standard RFC 4301*, Dec. 2005.

[17] W. Townsley et al., "Layer Two Tunneling Protocol "L2TP"," *IETF Standard RFC 2661*, Aug. 1999.

[18] B. Patel, B. Adoba, W. Dixon, G. Zorn, and S. Booth, "Securing L2TP using IPSec," *IETF Standard RFC 3193*, Nov. 2001.

[19] S. Redana, M. Lott, and A. Capone, "Performance Evaluation of Point-to-Multi-Point (PMP) and Mesh Air-Interface in IEEE Standard 802.16a," in *IEEE Vehicular Technology Conference*, 2004, pp. 3186-3190.

[20] E. Sakhaee and A. Jamalipour, "Stable Clustering and Communications in Pseudolinear Highly Mobile Ad Hoc Networks," *IEEE Transactions on Vehicular Technology*, vol. 57, no. 6, pp. 3769-3777, Nov. 2008.

[21] M. Sherman, K. McNeill, K. Conner, P. Khuu, and T. McNevin, "A PMP-Friendly MANET Networking Approach for WiMAX/IEEE 802.16," in *MILCOM 2006*, 2006, pp. 1-7.

[22] G. McGregor, "The PPP Internet Protocol Control Protocol (IPCP)," *IETF Standard RFC 1332*, May 1992.

[23] "C# Language Specification, 4th Edition," *Emca Standard EMCA-334*, Jun. 2006.

[24] "Common Language Infrastructure (CLI), 4th Edition," *Emca Standard EMCA-335*, Jun. 2006.

[25] F. Thiel and M. Krasnyansky. (2002) Universal TUN/TAP device driver. Linux Kernel 2.6.30.3 Source Code.

[26] ACIS Lab, University of Florida. (2009, Jul.) IPOP. [Online]. http://www.grid-appliance.org/wiki/index.php/IPOP

Vita Auctoris

Nicholas Charles Doyle was born in 1981 in Halifax, Nova Scotia. He graduated from Halifax West High School in 1999 and completed a B.Eng. at Dalhousie University in 2004. He then moved to Chatham, Ontario and worked for Siemens VDO Automotive for 4 years, earning his P.Eng. designation in 2008. He is currently an M.A.Sc. candidate for at the University of Windsor. He plans to graduate in the summer of 2009 and begin a Ph.D. in Engineering Science at Simon Fraser University in the fall of 2009.